



# **NAVAL POSTGRADUATE SCHOOL**

**MONTEREY, CALIFORNIA**

## **THESIS**

**IMPLEMENTATION OF INTEL VIRTUAL MACHINE  
EXTENSION ROOT OPERATION ON THE NPS LEAST  
PRIVILEGE SEPARATION KERNEL**

by

Jayce Martinsen

September 2010

Thesis Advisor:  
Second Reader:

Cynthia Irvine  
David Shifflett

**Approved for public release; distribution is unlimited**

THIS PAGE INTENTIONALLY LEFT BLANK

<b>REPORT DOCUMENTATION PAGE</b>			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
<b>1. AGENCY USE ONLY (Leave blank)</b>		<b>2. REPORT DATE</b> September 2010	<b>3. REPORT TYPE AND DATES COVERED</b> Master's Thesis	
<b>4. TITLE AND SUBTITLE</b> Implementation of Intel Virtual Machine Extension Root Operation on the NPS Least Privilege Separation Kernel			<b>5. FUNDING NUMBERS</b>	
<b>6. AUTHOR(S)</b> Jayce G. Martinsen			<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>	
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Naval Postgraduate School Monterey, CA 93943-5000				
<b>9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> National Science Foundation, grant No. DUE-0414012.			<b>10. SPONSORING/MONITORING AGENCY REPORT NUMBER</b>	
<b>11. SUPPLEMENTARY NOTES</b> The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.				
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> Approved for public release; distribution is unlimited			<b>12b. DISTRIBUTION CODE</b>	
<b>13. ABSTRACT (maximum 200 words)</b>  A virtual machine monitor (VMM) supports execution of multiple unmodified operating systems in virtual machines (VMs) on one computer. VMM support has been added to the Intel IA 32 architecture. Enforcement of data flow policies between VMs requires a highly trustworthy VMM. Such VMMs take advantage of hardware support. The work described here explores whether the Naval Postgraduate School Least Privilege Separation Kernel (LPSK) can incorporate Intel hardware support for virtualization.  The Intel documentation and LPSK code were reviewed to determine the changes required to transition the target processor to Virtual Machine Extension (VMX) root operation. First, paging in the LPSK had to be enabled. Requirements for the VMXON and VMXOFF instructions were determined and changes were made to the LPSK to enable the target processor to transition to VMX root operation.  Testing showed that the changes to the LPSK allowed the target processor to successfully transition to and from VMX root operation.				
<b>14. SUBJECT TERMS</b> LPSK, Virtual Machine, Intel Virtual Machine Extension.			<b>15. NUMBER OF PAGES</b> 79	
			<b>16. PRICE CODE</b>	
<b>17. SECURITY CLASSIFICATION OF REPORT</b> Unclassified	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b> Unclassified	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b> Unclassified	<b>20. LIMITATION OF ABSTRACT</b> UU	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)  
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release; distribution is unlimited**

**IMPLEMENTATION OF INTEL VIRTUAL MACHINE EXTENSION ROOT  
OPERATION ON THE NPS LEAST PRIVILEGE SEPARATION KERNEL**

Jayce G. Martinsen  
Civilian, Naval Postgraduate School  
B.S., Brigham Young University, 1994  
M.S., Arizona State University, 2002

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL  
September 2010**

Author: Jayce G. Martinsen

Approved by: Cynthia E. Irvine  
Thesis Advisor

David Shifflett  
Second Reader

Peter J. Denning  
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

## **ABSTRACT**

A virtual machine monitor (VMM) supports execution of multiple unmodified operating systems in virtual machines (VMs) on one computer. VMM support has been added to the Intel IA 32 architecture. Enforcement of data flow policies between VMs requires a highly trustworthy VMM. Such VMMs take advantage of hardware support. The work described here explores whether the Naval Postgraduate School Least Privilege Separation Kernel (LPSK) can incorporate Intel hardware support for virtualization.

The Intel documentation and LPSK code were reviewed to determine the changes required to transition the target processor to Virtual Machine Extension (VMX) root operation. First, paging in the LPSK had to be enabled. Requirements for the VMXON and VMXOFF instructions were determined and changes were made to the LPSK to enable the target processor to transition to VMX root operation.

Testing showed that the changes to the LPSK allowed the target processor to successfully transition to and from VMX root operation.

THIS PAGE INTENTIONALLY LEFT BLANK



## TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	BACKGROUND .....	1
B.	PURPOSE OF THESIS.....	2
C.	THESIS ORGANIZATION.....	3
II.	INTRODUCTION TO VIRTUAL MACHINES .....	5
A.	VIRTUAL MACHINES .....	5
B.	SECURE VIRTUAL MACHINE MONITORS.....	9
C.	INTEL-BASED VMM FOR HIGH SECURITY .....	12
D.	SEPARATION KERNELS .....	14
III.	IMPLEMENTATION .....	21
A.	INTRODUCTION.....	21
B.	SEGMENTATION AND PAGING.....	21
C.	VMXON.....	26
IV.	VIRTUAL MACHINE CONTROL STRUCTURE ANALYSIS .....	39
V.	CONCLUSIONS AND FUTURE WORK.....	43
A.	CONCLUSIONS .....	43
B.	FUTURE WORK.....	44
	APPENDIX A .....	47
	APPENDIX B .....	51
	APPENDIX C .....	57
	LIST OF REFERENCES .....	59
	INITIAL DISTRIBUTION LIST.....	63

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF FIGURES

Figure 1.	Type I VMM.....	8
Figure 2.	Type II VMM.....	9
Figure 3.	Protection rings. After [19] .....	11
Figure 4.	Ring compression. From [15] .....	12
Figure 5.	Segmentation with paging. From [31] .....	22
Figure 6.	Multiple segments. From [32] .....	23
Figure 7.	Control register 0. After [31] .....	24
Figure 8.	32-bit paging structures. After [32] .....	25
Figure 9.	Control register four (CR4). From [33] .....	25
Figure 10.	CPUID 0x0 return values.....	27
Figure 11.	CPUID 0x1- return values .....	28
Figure 12.	Values returned in ECX with CPUID called with a value of 0x1. From [33].....	28
Figure 13.	Values returned in EDX with CPUID called with a value of 0x1. From [36].....	29
Figure 14.	MSRs showing required values for CR0 .....	32
Figure 15.	MSRs showing CR4 required values .....	32
Figure 16.	VMXON pseudo code. From [44] .....	35
Figure 17.	Further explanation of VMXON GP() exceptions. From [44] .....	35
Figure 18.	VMXON parameter .....	36
Figure 19.	VMCS region format. From [43] .....	39
Figure 20.	VMCS component encoding structure. From [43] .....	40
Figure 21.	VM-entry/VM-exit.....	41

THIS PAGE INTENTIONALLY LEFT BLANK

## LIST OF TABLES

Table 1.	Trials and outcomes for testing VMXON.....	38
----------	--	----

THIS PAGE INTENTIONALLY LEFT BLANK

## **ACKNOWLEDGMENTS**

This material is based upon work supported by the National Science Foundation, under grant No. DUE-0414012. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author, and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK



# **I. INTRODUCTION**

## **A. BACKGROUND**

Software for the virtualization of computer hardware has existed for some time. This type of software allows multiple Operating Systems to operate on one physical CPU and is often called a Virtual Machine Monitor (VMM). IBM was working on developing the CP-67, which was one of the earliest versions of a virtual machine monitor, in the 1960s. A virtual machine monitor (VMM) is software that provides isolated duplicate copies of the actual machine environment for virtual machines to operate in. A virtual machine (VM) is software that operates within this duplicate machine environment provided to it by the VMM. Goldberg defines a virtual machine in the following way:

A hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode. [1]

A VMM is a control program that handles the accesses of the VMs to the real hardware of the system (i.e., memory, hard drives and I/O devices). VMMs have been classified into two categories, Type I and Type II. A Type I VMM is described as running directly on hardware, and a Type II VMM is loaded as an application on an existing host operating system. Because of the VMM's unique position of controlling the resources of the system that are provided to each VM, the VMM concept has been applied in the development of systems designed to separate mandatory security classes.

Software that is designed to separate the resources of a system into partitions and control access flows between these partitions is called a "separation kernel." Recently, the Information Assurance Directorate of the U.S. National Security Agency published a guideline for developing separation kernels for environments requiring high robustness. This guideline is called the Separation Kernel Protection Profile (SKPP) [2]. One of the requirements of this guideline is that the separation kernel implements the principle of least privilege (PoLP) to ensure subjects only have as much access to system resources as necessary for their intended functions.

As a part of the Trusted Computing Exemplar project, a group at the Naval Postgraduate School designed and prototyped a separation kernel to meet the requirements of the SKPP. This kernel implements the principle of least privilege and was termed a “least privilege separation kernel” (LPSK). Timothy Levine et al. described the LPSK’s implementation as follows:

Each subject is only given a minimum set of logically separated resources necessary to perform its assigned task, and the sharing of resources between subjects is rigorously controlled by the kernel. A separation kernel that correctly implements these properties can meet the objective to minimize and confine damage with a high level of assurance. [3]

## **B. PURPOSE OF THESIS**

Most VMMs in the past ran on large mainframe computers because commodity processors were not designed to be virtualized. To meet the need for a commodity processor that can be used for VMMs, Intel has developed processors with virtualization technology built in so that the Intel IA 32 architecture is virtualizable. Intel added instructions to the instruction set to handle virtualization. The extra instructions implemented on the Intel architecture, are called Virtual Machine Extensions (VMX). With these instructions handled in the hardware, the process of controlling virtualization can be simpler and faster. Under Intel’s virtualization environment, there are two modes of operation, VMX root operation and VMX non-root operation. Software controlling the virtual environment (i.e., the VMM) runs in VMX root operation and guest operating systems that run on top of the controlling software (i.e., the VMs) run in VMX non-root operation. The effort of this thesis will be to determine the requirements for booting and transitioning the LPSK into the VMX root operation, in order to allow the LPSK to operate as a VMM. Some of the benefits of allowing the LPSK to run in a virtualized environment are:

- The ability to run multiple Operating Systems that each can operate with different security attributes.
- Higher confidence in a VMM running multiple OSs that cannot communicate through covert channels.

- This system could potentially allow multiple OSs on one computer to operate in a non-system high environment.

## **C. THESIS ORGANIZATION**

The remainder of this thesis is organized as follows: Chapter II provides an introduction to virtual machines, separation kernels, and the least privilege separation kernel. Chapter III describes the implementation details of booting and transitioning the LPSK to VMX root operation on the Intel IA 32 architectures. Chapter IV gives a brief introduction to the virtual machine control structure (VMCS) and its purpose in the Intel IA 32 architecture. Chapter V describes the conclusions drawn from the efforts of this thesis and proposes possible future work that could be performed.

THIS PAGE INTENTIONALLY LEFT BLANK

## **II. INTRODUCTION TO VIRTUAL MACHINES**

### **A. VIRTUAL MACHINES**

The history of virtual machines reaches back to the early 1960s. They were originally developed to solve some specific problems that system programmers experienced when developing and testing operating systems for computers. First, system programmers could not test their changes to system programs without having full access to the system and its hardware. This meant that other users could not be using the system at the time the system programmer did tests. So, system programs were often tested in off hours, when the work would not interrupt that of other system users. Second, to test efficiencies of different designs of system programs, each change would have to be loaded and tested individually. To resolve these and other issues, system designers at IBM [4] in the early 1960s began to develop a means to partition a computer into smaller versions of itself called virtual machines. Although other companies were developing similar solutions, IBM's was one of the first to be commercialized. Goldberg defines a virtual machine as:

A hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode. [1]

A Virtual Machine Monitor (VMM) is a control program that provides the duplicate versions of the host machine to the Virtual Machines (VMs). The VMM will provide each VM with resources. In some cases, such as for portions of memory, these resources are for the exclusive use of a VM; whereas, in other cases, such as that of processor time, the resources are multiplexed among all VMs. Within these smaller versions (VMs) of the actual computer, resources are managed in a fashion similar to those of the original machine. Thus, the system programmer is able to test changes on a "virtual" image of the actual machine, and because this smaller "virtual machine" version of the real machine is logically isolated from the other virtual machines, if it crashes the

other virtual machines are unaffected. Partitioning a large system into smaller virtual images of itself allows efficient testing of differing design decisions of system programs.

In order to create multiple running simulated copies on a single host machine, IBM developed a construct it called the Control Program or CP for short. IBM also developed a simple single user operating system called the Conversational Monitor System (CMS) that could be loaded on top of the Control Program. The Control Program was responsible for interpreting the virtual instructions of the CMS and multiplexing the processor and storage between multiple copies of the CMS. The IBM 360 model 67 with CP/CMS loaded was the first widely available virtualization system [5]. IBM produced the 360 model 67 around 1967 to provide a platform for a next generation time sharing environment to universities and large corporations that allowed them to share computing resources between departments and users.

What IBM termed as the “Control Program” has become known as a virtual machine monitor (VMM). A virtual machine monitor is software that provides isolated simulated copies of the actual machine environment for the virtual machines to operate in and controls access of the VMs to the actual system resources. Goldberg defines VMMs as:

Software which transforms the single machine interface into the illusion of many. Each of these interfaces (virtual machines) is an efficient replica of the original computer system, complete with all the processor instructions (i.e., both privileged and non-privileged instructions) and system resources (i.e., memory and I/O devices). [6]

Goldberg made an effort to clearly define what can be classified as a virtual machine. One of the requirements was that it be efficient. His definition of efficient was that “a statistically dominant subset of the virtual processor’s instructions execute on the host processor in native mode” [1]. He explained that the performance of virtual machines is bounded between that of the real machine and a machine where all instructions are interpreted by software. In contrast to simulators and emulators, he reserves the term virtual machine for a system for which the subset of instructions that

are directly executed on the hardware is maximized. He further classifies virtual machine monitors into categories of Type I, Type II and Hybrid.

Because the concept of virtual machines was just emerging, many processors had not been built to support the special requirements of a virtual machine. The ultimate goal of a processor architecture designed to support the requirements of virtual machines is to provide an execution environment, such that software executing in the virtual machine cannot distinguish between the virtual and the real hardware environment. To create such an environment, it is necessary to ensure that the operation of one virtual machine does not affect that of any other virtual machine in operation. This means that the hardware state of the virtual processor must appear to be used exclusively by the particular VM and should not be affected by the execution of the other VMs. Goldberg described requirements that hardware had to provide in order to be able to support a virtual machine.

- 1) Non-privileged instructions must be executed in nearly the same manner in both privileged and non-privileged states on the processor.
- 2) A means of protecting the space of the VMM and active VMs from each other such as address translation (i.e., segmentation or paging).
- 3) A means must be provided for notifying the VMM when a virtual machine attempts sensitive instructions. This means of notifying or trapping must not cause unrecoverable errors. And it must be possible for the VMM to simulate the effect of the instruction. Sensitive instructions include:
  - a. Instructions that change the state of the machine. (i.e., privileged and non-privileged states).
  - b. Instructions that read or write to sensitive system registers.
  - c. Instructions that reference mechanisms that the VMM uses to create or manage the VMs such as, the memory system, or storage protection mechanisms.
  - d. All I/O instructions.

A Type I VMM is described as running directly on hardware. It is kernel-level software developed with the means to support virtual machines. This software must be able to schedule and allocate CPU, memory, I/O and storage resources of the physical machine. A Type I VMM executes directly on the hardware as in Figure 1. Unmodified guest operating systems execute at the VMM interface. In order for a machine to be able to support a Type I VMM, it must have means to support each of the hardware requirements listed above. IBM System/370 [7], its descendant z/VM [8], and Microsoft's Hyper-V server [9] are examples of a Type I VMM.

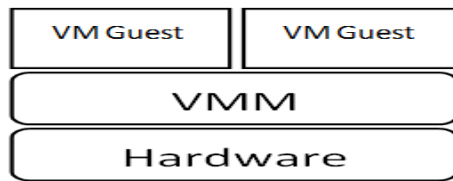


Figure 1. Type I VMM

A Type II VMM is loaded as an application on an existing host operating system. The Type II VMM provides virtualization support but relies on the host operating system to handle the scheduling and allocation of system resources. A Type II VMM is generally depicted with the host operating system located closest to the hardware, as in Figure 2. To support a Type II VMM, hardware must support the same rules stated above, but there must be additional requirements to ensure the host OS does nothing to invalidate the rules above. First, the host OS must not invalidate the requirement that the non-privileged instructions execute roughly the same in privileged and non-privileged modes. Second, the host OS must have the means to protect the VMM from other VMs and processes. Third, the host OS must have a means of signaling the VMM when a VM attempts to execute a sensitive instruction. VMware Workstation [10] is a modern example of a Type II VMM. VMware Workstation operates as an application on either a Linux or Windows operating system and will host guest VMs that support the underlying CPU architecture.



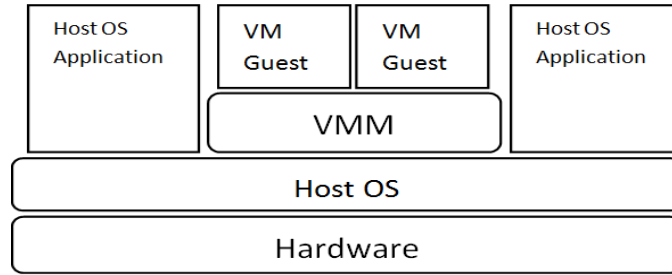


Figure 2. Type II VMM

A hybrid VMM is described as a VM where all supervisor or privileged instructions are interpreted. This type of VMM was developed mainly as a simple way to overcome implementation difficulties created by hardware that was not designed to support virtualization.

It is common in the literature to see the term *hypervisor* used interchangeably with VMM. The term hypervisor may have come from the IBM/370's use of the term hypervisor-call to refer to a call a guest operating system could use to access services from a more privileged control program.

## B. SECURE VIRTUAL MACHINE MONITORS

In the Department of Defense and intelligence communities, the ability to ensure that classified material is not leaked to a lower security level is of profound importance. In a multilevel security environment, the flow of information that is allowed is determined by a security policy. The security policy is often described in an access matrix in which subjects (users, processes) are allowed various modes of access to an object (files, memory) through access rights found in the subject-object entry of the matrix. To enforce the policy, the system must ensure that subjects only have access to those objects for which they are allowed in the access matrix and all other flows of information between subjects are not possible. As Lampson [11] points out, there are quite a few channels that allow flow of information on shared systems; these must be considered to ensure correct policy enforcement. There are also nontraditional means of information flow on shared systems. Among these are covert channels. A covert channel

is created when system mechanisms not intended to cause information to flow, are used to transfer information from one security domain to another in violation of policy. In a multilevel system, it is essential that the system has the ability to control the flow of information and that unintended means of information flow have been identified and mitigated to ensure that information does not leak from one security level to another in violation of the intended security policy.

The ability of a virtual machine monitor to create separate isolated environments on one physical machine has interested researchers and developers of secure computing systems through the years. Madnick and Donovan [12] suggested the benefits of using VMMs for security and Rhode [13] was the first to present the idea of using VMMs as security kernels. In the late 1970s, Systems Development Corp. built a security kernel implementation of IBM's VM/370 called KVM/370. [14] In the late 1980s, Digital Equipment Corp. created a VMM security kernel for its VAX architecture. [15] This security kernel supported multiple virtual machines on a single VAX system and provided isolation and controlled sharing of sensitive data. This security kernel applies both mandatory and discretionary access controls to virtual machines based on the Bell and LaPadula confidentiality [16] and Biba [17] integrity models, respectively. It also applies access control lists on all objects. In the VAX security kernel, there are two types of subjects: users and virtual machines. Objects in this security kernel include virtual machines and virtual disks.

To create a secure VMM, it is first necessary to make sure that any instructions that reveal or allow access to sensitive items such as memory page tables or processor state will trap so that the VMM security kernel can emulate their behavior. When evaluating the VAX architecture against Goldberg's requirements it was found that many of the instructions were sensitive and non-privileged and so required changes to the hardware architecture to allow these instructions to trap to the VMM. A second hurdle that had to be surmounted was the ring mapping. Hierarchical protection domains [18] are often called *protection rings* or *privilege rings*. These rings were designed to provide different levels of resource access for different modes of operation within a process. The kernel code of the operating system may be given access to areas of memory or registers

that application code would not be allowed access to. Often, in systems that use privilege rings, there are 4 privilege rings labeled zero to three, as shown in Figure 3.

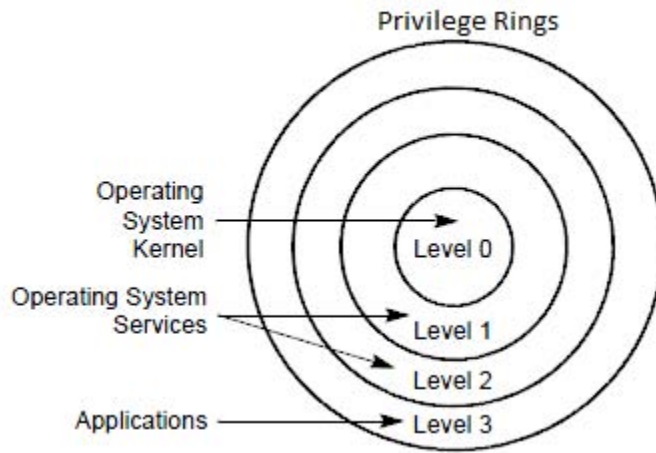


Figure 3. Protection rings. After [19]

The rings are organized in order from most privileged at the lowest level (ring 0) to the least privileged at the highest level. These privilege rings are a means for the OS to protect itself from faulty or malicious code of the applications it runs. The OS can allow different levels of access to system resources by running applications at different privilege rings. The level of access is enforced by the hardware. Gates, or call instructions, are provided to allow less privileged applications to access resources in more privileged rings in a controlled fashion. In most commodity OSs, only two privilege rings are used. The VAX architecture required the use of all four rings. The VAX team came up with an approach they called “ring compression.” Essentially, the VAX team ran the VM kernel and VM executive in the same privilege ring. Figure 4 shows how the protection rings of the virtual processor are mapped to the protection rings of the actual VAX processor. The VAX team states that though with this approach the virtual machine’s kernel mode is no longer protected from its executive mode, it was not a problem for the two OSs of interest to the VAX security kernel—and that compressing the VM kernel and VM executive in this way did not compromise the security of the VMM

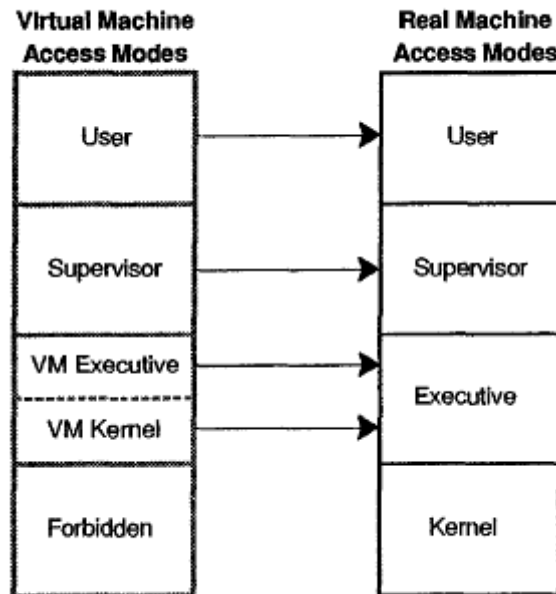


Figure 4. Ring compression. From [15]

The VAX system also had to tackle another difficult area for VMM security kernels: the I/O interface. They created a specialized kernel call mechanism that ensured all I/O access trapped to the security kernel in a highly efficient manner.

### C. INTEL-BASED VMM FOR HIGH SECURITY

In 2000, problems associated with implementing secure VMMs on Intel commodity processors were investigated [19]. The Intel platform's 250 instructions were assessed against Goldberg's requirements for virtualizability. Seventeen instructions were found to be sensitive and unprivileged, leading to the conclusion that Intel's current architecture should not be used as a development platform for a secure virtual machine monitor because it could not ensure secure isolation of classified and unclassified virtual machines. Another problem with developing a secure virtual machine on this architecture is that the code required to implement the VMM becomes large, due to the need to provide binary translation of the sensitive and unprivileged instructions of the architecture. The size of the VMM kernel then becomes more difficult to evaluate

against high assurance requirements that call for minimization of the kernel. The benefits that a highly secure Type I VMM built on the Intel platform would provide if the architecture could support it were pointed out:

First, a Type I VMM can provide a high degree of isolation between VMs. Second, existing popular commercial operating systems for the processor and their applications can be run in this highly secure environment without modification. [20]

Intel has recently added hardware virtualization support that allows the architecture to meet Goldberg's virtualization requirements.

A more recent example of secure VMMs can be found in the Terra project, where a VMM termed a *trusted VMM* (TVMM) was created [20]. This VMM is considered trusted because it uses tamper-resistant hardware called the trusted platform module (TPM) to certify the firmware (BIOS), system boot loader, and the VMM. The TPM is a crypto-coprocessor that has a private key and can generate SHA-1 hashes [21]. A hash can be generated for each piece of loaded code to ensure the code is an exact copy of what the system expected to load. The Terra project was built on commodity hardware with the intent to be able to run commodity operating systems. In order to provide a "trusted" or secure path for users to communicate with the TVMM, Terra developers found several areas that would benefit from hardware support. First, because commodity I/O device drivers could not be trusted, a means of creating a trusted path between the TVMM and devices, such as the keyboard, mouse, and video card, was needed. One means of accomplishing this was to encrypt all communications with these devices, but this would have required moving the device drivers into the TVMM. As an alternative, Terra developers split the device interfaces. A video card driver could split the 2D and 3D interfaces to allow the simpler 2D interface to be implemented in the TVMM and the more complicated 3D interface could be allowed direct access by the VMs. Second, a secure counter that would only increment would improve security by providing protection from file system rollback attacks. Third, Terra developers found that hardware support for controlling access to devices such as the DMA controller and PCI bus would allow the TVMM to not be required to trust the commodity drivers for these devices. Although the Terra TVMM may be ensuring that the VMM booted has not been modified

there is no discussion about whether a VMM can be considered trustworthy. In addition, there is no evidence that Terra could be trusted to enforce a mandatory policy.

As anticipated, hardware support for virtual machines on commodity processors is now beginning to be implemented in current Intel and AMD processors. This hardware support for virtualization on commodity processors will likely extend the benefits of secure VMMs beyond the mainframe environments of finance, telecommunications and defense.

#### **D. SEPARATION KERNELS**

In 1981, J.M. Rushby wrote, “It is widely recognized that VMMs provide a suitable basis for the construction of secure systems” [22]. Rushby pointed out that one of the Achilles heels of the security kernels was that these kernels were designed and formally reviewed to ensure their mechanisms rigidly apply a multiple level security (MLS) policy, and yet to provide the practical functionality required in a typical user environment, multilevel processes would need to be created. An example of this was a simple print spooler. In order to get print spooling functionality, the spooler would have to become a ‘trusted process’ and be allowed access to spool files at a range of security levels, i.e., it is a multilevel process. Rushby claims that, “The combination of a security kernel and trusted processes is hard to understand and even harder to verify because it does not represent a separation of concerns but a confusion of the same: neither member of the combination is independent of the properties of the other” [23]. He also pointed out that a formal model for a system comprised of this combination of security kernel and trusted processes was not available.

Rushby proposed that separating specialized components and services and reviewing their security properties individually would allow for a cleaner formal proof of a system’s security properties. To create this separation, he suggested a model similar to that of a distributed system. And, to make this system cost effective, he suggested implementing it on a single processor. To accomplish the implementation of a distributed system on a single system’s hardware, the physical security devices used in a distributed system would need to be implemented logically. He pointed out that, for this

architecture, the security kernel's purpose is to create this distributed environment and that the responsibility for security policy enforcement is that of the individual components and no longer centralized in the kernel itself.

It becomes evident that the kernel component of this model resembles what we have described above as a virtual machine monitor. Indeed, Rushby points out that,

In an ideal, physically distributed implementation, each component of the system runs on its own private and physically isolated machine. The task of a security kernel, therefore, is to provide an isolated 'Virtual Machine' (VM) for each component and to handle communications between these virtual machines. A kernel of this form is obviously very similar to a 'Virtual Machine Monitor' (VMM). [23]

Rushby felt that there were significant differences between current implementations of the VMM concept and his proposed security kernel. Specifically, he points out that his concept does not require that the VMM, "provide exact copies of the base hardware (or even for all the VMs to be alike)—but there is a requirement for it to provide communication channels between some of its VMs" [23]. So, to avoid confusion with the current definition of VMMs he decided to call his concept a 'separation kernel.'

Separation means that one process cannot intentionally or unintentionally influence another process in ways that were not intended by the system. As stated by Martin et al., "Any means for one process to disturb another, be it by communication primitives, by sharing of data, or by subtle uses of kernel primitives not intended for communication, is ruled out when two processes are separated" [23]. Again, this separation does not imply that there can be no communication between processes, only that it is verifiable that the only flow of information is by channels intended by the system. So, a separation kernel's purpose is to create, as Rushby stated, "an environment which is indistinguishable from that provided by a physically distributed system." [23].

Separation kernels may provide a foundation for the creation of multilevel secure systems (MLS). Recently the Information Assurance Directorate of the U.S. National Security Agency published a guideline for developing separation kernels for

environments requiring high robustness. This guideline is called the Separation Kernel Protection Profile (SKPP) [2]. This profile states some of the separation kernel's core functionality expected as follows:

- Protection of all resources including the CPU, memory, storage devices etc. from unauthorized access.
- Separation of resources used for security from the resources provided to subjects.
- Partitioning and isolation of resources provided to subjects.
- Control of information flows between partitions and resources provided to partitions.
- Implementation of the principle of least privilege (PoLP) to ensure subjects only have as much access to system resources as necessary for their intended functions.

The last requirement, i.e., to implement the principle of least privilege, is often overlooked in separation kernel design due to the view that separation kernels are solely responsible for isolation of processes. While isolation is a highly desirable quality of a separation kernel, use models often require that some communication between partitions be allowed. As Irvine et al. state, "In practice, however, separation kernels are often used to share hardware among kindred activities that have reason to communicate in some controllable fashion" [24]. Allowing some controlled communication between partitions is often desirable even in secure applications. To meet real world needs, separation kernels often provide controlled communication channels between partitions. The problem lies in the fact that traditionally separation kernels were only concerned with secure separation of partitions, rather than the controlled sharing of resources between partitions. This can be solved in two ways. First, by creating a special trusted partition that mediates all communication between partitions according to policy. Second, by applying the principle of least privilege in a way that permits controlled sharing among partitions in accordance with the overall security policy. In an effort to address this issue, Irvine et al. describe a Least Privilege Separation Kernel (LPSK) model that "builds on the traditional separation abstraction by extending the granularity of described elements



to the subjects and resources within the partition” [25]. This granularity should be at the same level as the resources exported by the separation kernel.

With the granularity of control at the same level as the resources exported, the separation kernel can control access to each resource within each partition. This allows the LPSK to give a subject access to an individual resource within a partition while limiting access to other resources within that partition.

The Trusted Computing Exemplar Project (TCX) [25] is intended to provide an open example of a high assurance trusted computing development process. To demonstrate the high assurance development process, a high assurance micro kernel is being created. This kernel is being developed to meet the requirements of the SKPP [2] and for evaluation at evaluation assurance level seven (EAL7) [26]. The kernel has been designed and implemented as a least privilege separation kernel. The effort of this thesis will be to implement Intel hardware support for virtualization in the TCX LPSK. The TCX LPSK could then be used to develop a highly secure VMM and reap the benefits described by Robin and Irvine [20].

In early 1999, VMware developed techniques to work around the three concerns that Popek and Goldberg [27] determined made the Intel architecture non-virtualizable. They used binary translation to rewrite instructions that were both sensitive and unprivileged, data structures called ‘shadow page tables’ to emulate the MMU, and emulation of I/O devices to make the x86 architecture virtualizable. VMware’s product was developed as a Type II hypervisor to operate on a host workstation OS.

Most commodity OSs operate in what is called *protected mode*. Protected mode is an x86 operating mode [28] and was designed to provide hardware, support for mechanisms, such as virtual memory, and paging that help protect the OS against applications it is running. On an x86 processor that supports protected mode, the system will boot in real mode to maintain backward compatibility. Protected mode can only be entered after the system initialization module sets up descriptor tables such as the global descriptor table (GDT) and enables the protection enable (PE) bit in control register zero (CR0). The software must also set the 21st address line (A20) bit. The A20 bit is

originally disabled to allow the system to boot in compatibility mode. When this bit is disabled, the 21st address line is not used allowing the system to access at most  $2^{20}$  bytes for a total of 1 megabyte of memory. Enabling this bit in protected mode allows the use of all the address lines of the system and access to memory beyond 1 megabyte.

A feature of protected mode, which protects the kernel from its less privileged applications, is the use of hierarchical protection domains [18]. As discussed earlier, hierarchical protection domains are often called protection rings or privilege rings. Intel documentation calls them hardware privilege levels (PLs). In most commodity OSs, only two privilege rings are used. The OS operates at PL 0 (most privileged) and applications operate at PL 3. This two privilege level approach creates an issue when trying to virtualize these systems. The guest OS must be protected from its own applications but must still be prevented from accessing resources that must be controlled by the VMM. This problem has been addressed in a variety of ways by various vendors.

To address this issue, VMware used the protection mechanisms (paging and segmentation) of privilege rings to ‘de-privilege’ the guest OS. VMware’s approach creates a Type II VMM, but is efficient enough that VMware became popular.

Xen [29] developers preferred to make changes to each guest operating system, so that instead of executing sensitive unprivileged instructions in the guest OS, a call to VMM services is invoked. This approach is commonly called *paravirtualization*. Paravirtualization side steps the whole issue of sensitive and unprivileged instructions not trapping to the VMM, but takes on the added burden of rewriting areas of the guest VM’s code to ensure they do not use these instructions.

Intel developed virtualization technology (VT) with the design goal of eliminating the need for paravirtualization and binary translation. This technology made it easier to implement VMMs by addressing: sensitive instructions to ensure that they trap and thus eliminating the need for binary translation. Intel VT created two new modes of processor operation called *VMX root operation* and *VMX non-root operation*. The VMM executes in VMX root operation and the VMs execute in VMX non-root operation. Both modes of

operation support all of the privilege rings. This allows guest operating systems to operate at PL 0 as they were originally designed. Intel VT allows the VMM to be simpler and allows support for unmodified OSs.

Intel virtual-machine extensions (VMX) are new instructions added to the Intel 64 and IA-32 architecture to support virtualization of the processor and system. VMX includes five instructions that manage VMX operation and five instructions that manage the virtual machine control structure (VMCS). These instructions are listed below and descriptions reflect the Intel documentation [30]:

VMX management instructions:

- VMXON- This enters the logical processor into VMX root operation.
- VMXOFF- This instruction will cause the processor to leave VMX operation.
- VMLAUNCH- This instruction launches a virtual machine with management data in the VMCS.
- VMRESUME- Resumes a virtual machine managed by the current VMCS.
- VMCALL- This instruction allows a guest VM to call the VMM for service. A VM exit occurs, transferring control to the VMM.

VMCS-maintenance instructions:

- VMCLEAR- This instruction sets the launch state of the VMCS, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area.
- VMPTRLD- This instruction makes the referenced VMCS active and current, loading the current-VMCS pointer with the address of the current VMCS based on the contents of the VMCS-data area.
- VMWRITE- This instruction writes a component to the VMCS.
- VMREAD- This instruction reads a component from the VMCS.
- VMPTRST- The current-VMCS pointer is stored into the destination operand.

THIS PAGE INTENTIONALLY LEFT BLANK

### **III. IMPLEMENTATION**

#### **A. INTRODUCTION**

A study of the Intel Software Developer's Manual showed that a processor must be in protected mode with paging enabled in order to operate in VMX root operation. As the LPSK prototype did not implement paging, it was first necessary to set up the paging structures and set the control register settings that were required to enable paging for the LPSK. Once paging was enabled, it was necessary to prepare the system to execute the VMXON instruction to transition the processor to VMX root operation. This section details the efforts required to set up the system for both paging and the execution of the VMXON instruction.

#### **B. SEGMENTATION AND PAGING**

In the IA-32 architecture, memory protection and management are provided by the use of segmentation and paging. Segmentation provides a means of keeping the code, data and stack of different processes separate. This allows multiple programs to run on the same system without interfering with each other's memory space. In addition, for high assurance systems, segmentation provides a mechanism that can be simply and elegantly mapped to a formal model. Figure 5 shows how segmentation is used to separate the linear address space of the processor into protected segments.

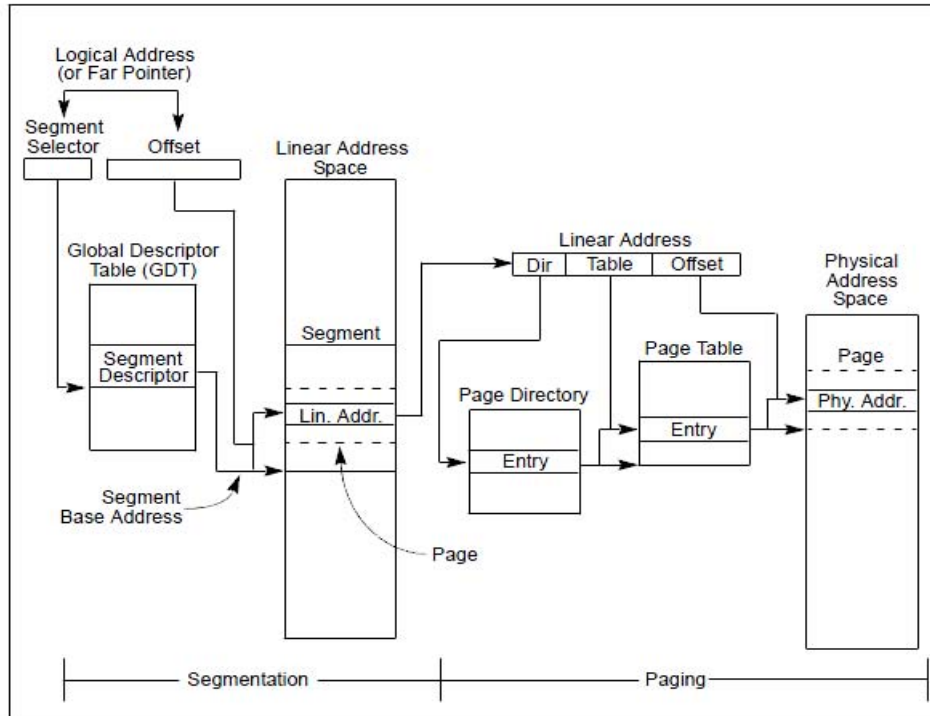


Figure 5. Segmentation with paging. From [31]

As Figure 5 shows, paging can also be used in conjunction with segmentation. Paging takes the linear address space and divides it into equal size pages. These linear address pages are then mapped to physical page frames using the paging data structures, as shown in the figure. Bits in the paging structures (page directory table and page table) can be used to control read and write access to individual page frames in memory.

Segments have access permissions and limits associated with them, as shown in Figure 6. When multiple programs are run the program's code, data and stack segments are generally assigned to segments addressed by an appropriately named segment register: code segment (CS), stack segment (SS) and data segment (DS). The hardware will enforce proper access to and the boundaries of each segment by checking the access and limit for each memory access made. Using local descriptor tables, each process on the system can be given its own segments and the hardware can then ensure that one process cannot read or write to another program's memory space using the mechanisms provided by the segmentation structures.

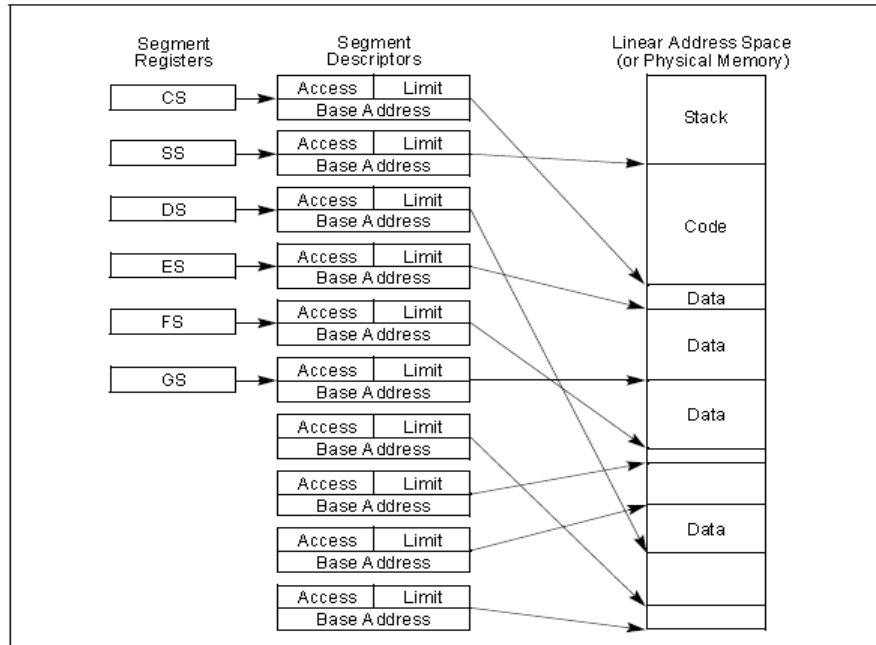


Figure 6. Multiple segments. From [32]

In order to get the processor in VMX root, operation system software must prepare the system to execute the VMXON instruction. The first requirement for executing the VMXON instruction is that the system be in protected mode with paging enabled. In protected mode segmentation is required but paging is optional. The LPSK prototype was implemented using segmentation to provide isolation of processes but did not implement paging. When segmentation is used without paging, the linear address space of the processor is mapped into the physical address space directly. Thus, memory management skips the page directory and page table structures when segmentation is enabled without paging. In order to transition the processor to VMX root operation, it was necessary to implement paging.

The Intel IA 32 architecture provides for three different modes of paging. The three modes of paging are 32-bit paging, Physical Address Extension (PAE) paging, and IA-32e paging. Software can determine the support a processor provides for different paging features using the CPUID instruction. For the LPSK, a 32-bit paging mode was chosen as it used 32 bit addressing. Paging is enabled by setting the paging bit in control register zero (CR0). Control register zero is a 32 bit or 64 bit register (the register will be

64 bit only on an x86-64 processor in long mode). The bits of the register act as control flags that can be set to modify the operation of the processor. Figure 7 shows the bits or control flags of control register zero. In order to enable paging on the processor it is necessary to set the PG bit (or bit 31) of CR0. In the Intel Software Developers Manuals it is common to see a reference to the 31 bit (or PG bit) of control register zero written as CR0.PG. This format for specifying specific bits of a register will be used in the remainder of this document.

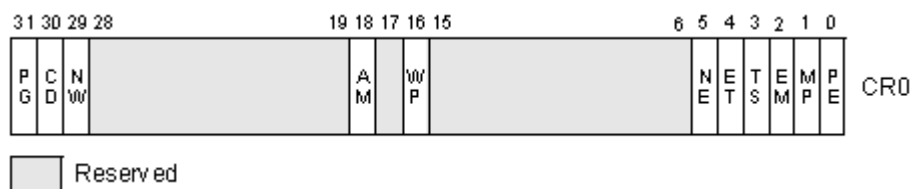


Figure 7. Control register 0. After [31]

Before setting the CR0.PG bit and enabling paging on the processor, software must first set up the data structures necessary to support paging. In 32-bit paging mode, linear addresses are translated using the page directory table and the page tables. All paging structures in the IA 32 architecture are 4KB in size. Some work had already been done on a function to set up the paging structures necessary to support paging in the LPSK. In particular, the `setup_paging()` function shown in Appendix A already had the code necessary to setup a 4KB aligned 4KB region in memory to be used as the page directory table. The function also set up a total of 1024 regions, each 4KB in size, to be used as the page table structures. Once created, descriptors for each of these structures are then added to the Global Descriptor Table (GDT).

In 32 bit paging mode, each paging structure contains 1024 32 bit entries. Each paging structure entry contains a physical address, which either points to another paging structure or a page frame as shown in Figure 8.



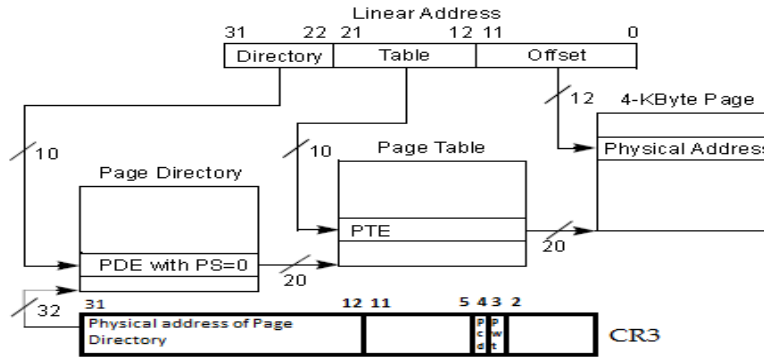


Figure 8. 32-bit paging structures. After [32]

Each entry in the page directory needs to contain the address of a page table that it references. This is accomplished with a for-loop to set bits 31:12 to the physical address of the page table referenced by each entry. Each entry has access control bits that must be appropriately set as well. The page table entries then must contain the physical address of the 4KB page referenced by the entry and have the appropriate access control bits set.

Once the paging structures are created and linked, paging can be enabled by setting the necessary bits in the control registers. First, control register three (CR3) bits 31:12 are set with the address of the page directory structure. The system uses this control register to access the first paging structure when translating a linear address. Second, 32 bit paging uses CR0.WP (write protect), CR4.PGE (page global enable), and CR4.PSE (page size extension). Figure 9 shows the layout of control register four (CR4).

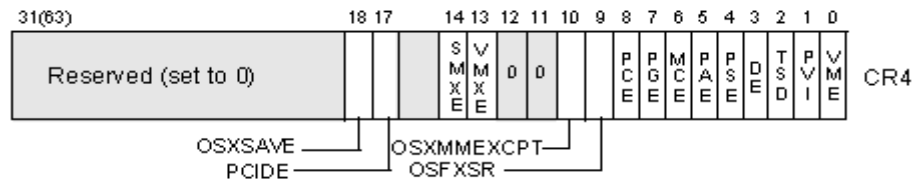


Figure 9. Control register four (CR4). From [33]

The write protect bit of CR0, when cleared, will allow supervisor level processes to write to read-only pages. For the LPSK this bit is set, thus inhibiting supervisor level processes from writing to read-only pages. The PGE bit of CR4 allows frequently shared pages to be marked as global to all users. While, the PSE bit allows the use of 4MB pages when set. When the PSE bit is clear, page sizes are restricted to 4KB. Both PGE and PSE were cleared to allow 4KB paging without global sharing of pages. Finally, to enable 32 bit paging, the page bit in CR0 must be set to one ( $CR0.PG = 1$ ) and the physical address extension of CR4 must be set to zero ( $CR4.PAE = 0$ ). Since paging cannot be enabled unless protected mode is enabled, the system must also check to ensure that the protection enable (bit 0) of control register zero is set ( $CR0.PE = 1$ ).

With the paging structures created, entries properly set and control registers set, it was assumed that paging would operate perfectly when the `setup_paging()` function was called. Unfortunately, this was not the case. Calling the `setup_paging()` function caused the system to reboot. The first thought was that paging was being called before the GDT had been initialized. This was ruled out by checking to ensure that the GDT had already been set up. It was determined that the Task-State Segments, which define the state of a execution environment for a task, including the base address of the first paging structure, were not being set. When the TSS section of the LPSK was written the code did not update the first TSS with the value of the physical address of the first paging structure from control register three (CR3). Once the code was changed (line 1004 of `tcx\trunk\kernel\kernel_ini2.c`) to set the address from CR3 for the first task, paging worked properly.

### **C. VMXON**

With the processor running in protected mode with paging enabled, a function to prepare the system to call the Intel IA 32 instruction VMXON was created. This function was named `setup_vmxon()` and was added to `tcx\trunk\kernel\kernel_ini2.c` and is shown in Appendix B. Before any other setup work could be performed, it was first necessary to check for the processor-specific hardware support to be provided for the VMM. This was accomplished using the Intel IA 32 instruction, `CPUID`. This instruction was introduced

in the early 1990s to allow programmers a means of determining a processor's make and model. Before this instruction was available, programmers had to write code to exploit minor differences between processors to determine a CPU's make and model [31]. The LPSK did not provide a function for calling CPUID so a function named `get_cpuid()` was created. The Intel Software Developer's Manual states the following in describing the CPUID instruction, "The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well)" [32]. This implies that the CPUID instruction takes no parameters as it explicitly expects its calling parameters to be provided in the EAX and ECX registers.

As a primitive test of the functionality of `get_cpuid()`, it was called with a parameter of 0x0 in the EAX register. The processor returned the following values, as shown in Figure 10.

EBX: 0x756e6547 =	ueG
ECX: 0x6c65746e =	letn
EDX: 0x49656e69 =	Ieni

Figure 10. CPUID 0x0 return values

The vendor ID string (GenuineIntel) was spelled out in the values returned in registers EBX, ECX and EDX, which is the expected value for the processor being used for testing. For the purposes of determining the support that the processor provides for VMX, CPUID is called with a value of 0x1 in the EAX register, which returns model and family information for the processor in EAX and feature-bits in ECX and EDX. The values returned for the test processor are shown in Figure 11.

EAX: 0x10676 =	0001 0000 0110 0111 0110	15 5 0 ← Bit count ← Bits
ECX: 0x8e3fd =	1000 1110 0011 1111 1101	15 5 0
EDX: 0xbfebfbbf =	1011 1111 1011 1111 1111	15 5 0

Figure 11. CPUID 0x1- return values

The values returned in ECX and EDX are interpreted using the Intel Software Developers Manual [36], as shown in Figure 12 and Figure 13, respectively.

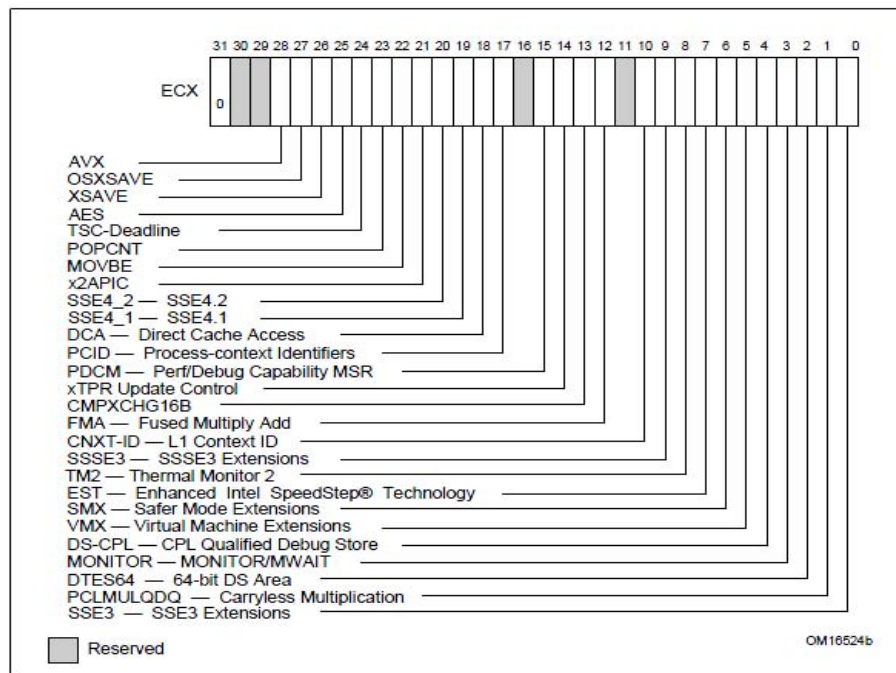


Figure 12. Values returned in ECX with CPUID called with a value of 0x1. From [33]

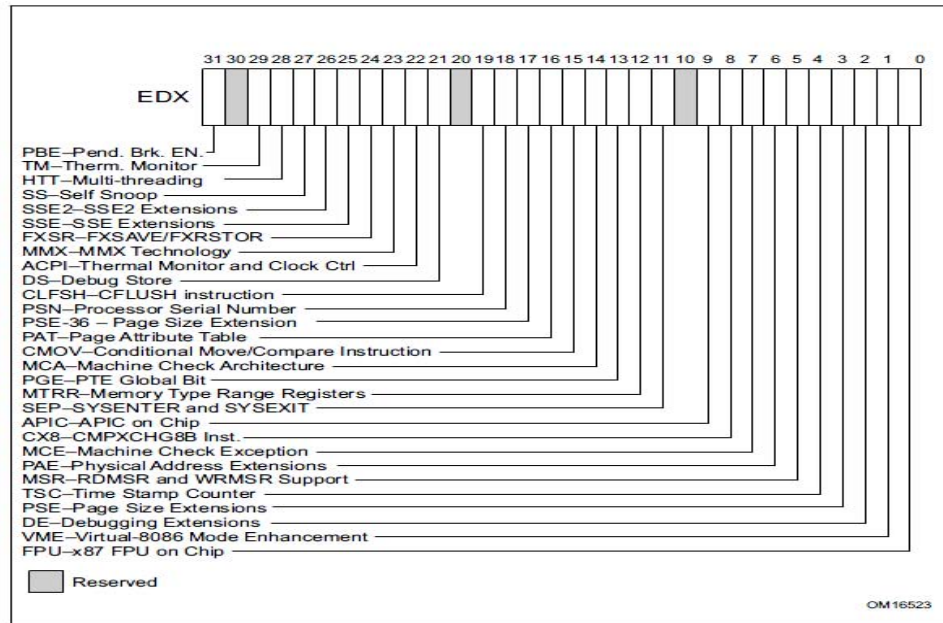


Figure 13. Values returned in EDX with CUID called with a value of 0x1.  
From [36]

If bit five in ECX is set (i.e., bit five equals 1), then the processor provides support for the Intel virtual machine extension instructions (VMX). If VMX instructions are not supported, then the `setup_vmxon()` function halts the processor. After checking that the processor supports the VMX instructions, the next step was to check whether the processor supports model-specific registers (MSRs). This can be accomplished by checking bit 5 of EDX. If this bit is set, then MSRs are supported by the processor.

MSRs are read to determine the VMX capabilities of the processor. VMX-specific MSRs are indexed starting at address 0x480 (Hex 480). MSRs are read using the Read from Model Specific Register (RDMSR) instruction. A function for calling the RDMSR instruction was not provided in the LPSK, so a function named `get_msr()` was added. The parameter (the address of a model specific register) to this function is placed in the ECX register. RDMSR returns the contents of the 64-bit MSR specified by the ECX register. The high-order bits of this register are loaded into the EDX register and the low-order bits are loaded into the EAX register. Constants were created for each of the MSRs used in the `setup_vmxon()` function. The naming of these constants was designed to match the name of each MSR as it appears in the Intel Software Developer's

Manual so that it is clear which MSR is being referenced (i.e., which value is being placed in the ECX register) with each call to `get_msr()`.

With the ability to read the MSRs, the system can now determine the processor's requirements for the size and initialization value necessary to set up two control structures that are required for VMX operation. The first of these control structures is the VMXON region. This region is defined in the Intel Software Developer's Manual as "a naturally aligned 4-KByte region of memory that a logical processor may use to support VMX operation" [33]. The second structure is called the Virtual Machine Control Structure (VMCS) and will be discussed in a later chapter.

To determine the processor-specific requirements for the VMXON region, the `IA_32_BASIC` MSR must be read. Bits 44 through 32 of this MSR determine the size in bytes that should be allocated for the VMXON region. The Intel manuals use the notation 44:32 to describe a continuous run of bits starting at 32 and continuing to bit 44. This notation will be used in the remainder of this document. The `setup_vmxon()` function reads the `IA_32_BASIC` MSR and uses the value returned (bits 44:32) to set a variable appropriately named `region_size`. Then a memory allocation function provided in the LPSK is called with `region_size` as a parameter. The memory allocation function returns the physical address of a 4KB aligned region in memory that has a size equal to `region_size`. The only initialization of this region that is required is to set bits 31:0 equal to the value of the VMCS revision identifier. This value is reported in the low-order (31:0) bits of the `IA_32_BASIC` MSR.

Certain VMX controls are reserved and must be set to either one or zero. This required value is called the control's *default setting*. MSRs are used to allow software to determine the default setting of a reserved control. The default settings for reserved controls can change if new functionality is added in newer processors, so it is important for software to determine the default settings for the processor on which it is running. The default setting for a reserved control can be one of three possible types [34].

- **Always-Flexible**-These have never been reserved.
- **Default0**-These are (or have been) reserved with default setting 0.
- **Default1**-They are (or have been) reserved with a default setting of 1.

The minimal set of control registers that must have their default values set to enable VMX root operation on a processor are CR0, CR4, and the IA\_32\_FEATURE\_CONTROL MSR. The CR0.PE and CR0.PG bits were set previously when setting up paging, but it is necessary to check the MSRs specific to this register to determine whether there are any other bits that must be set for the specific processor model. The two MSRs related to determining the required default setting of CR0 are, IA32\_VMX\_CR0\_FIXED0 and IA32\_VMX\_CR0\_FIXED1 MSRs. The logic for determining how these two MSRs can be used to find the default value for CR0 reserved bits is explained as follows:

They report on bits in CR0 that are allowed to be 0 and to be 1, respectively, in VMX operation. If bit X is 1 in IA32\_VMX\_CR0\_FIXED0, then that bit of CR0 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32\_VMX\_CR0\_FIXED1, then that bit of CR0 is fixed to 0 in VMX operation. It is always the case that if bit X is 1 in IA32\_VMX\_CR0\_FIXED0, then that bit is also 1 in IA32\_VMX\_CR0\_FIXED1; if bit X is 0 in IA32\_VMX\_CR0\_FIXED1, then that bit is also 0 in IA32\_VMX\_CR0\_FIXED0. Thus, each bit in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32\_VMX\_CR0\_FIXED0 and 1 in IA32\_VMX\_CR0\_FIXED1). [35]

The logic reduces to: if a bit is 1 in IA32\_VMX\_CR0\_FIXED0 then this bit in CR0 must be set to 1, and if a bit is 0 in IA32\_VMX\_CR0\_FIXED1 then that bit in CR0 must be set to 0.

The return values for IA32\_VMX\_CR0\_FIXED0 and IA32\_VMX\_CR0\_FIXED1 for the target processor in this work are shown in Figure 14.

IA32_VMX_CR0_FIXED0- EAX: 0x80000021	31	15	5	0
	1000	0000	0000	0000 0000 0010 0001
IA32_VMX_CR0_FIXED1- EAX: 0xffffffff	31	15	5	0
	1111	1111	1111	1111 1111 1111 1111
Required values for CR0-	1		1	1

Figure 14. MSRs showing required values for CR0

Although it was already known that CR0.PG (bit 31) and CR0.PE (bit 0) are required to be set, reading the MSRs for this register showed that it was necessary to ensure CR0.NE (bit 5) was set as well. The `setup_vmxon()` function reads the CR0 MSRs and ensures that CR0 is set according to the default settings required by the processor.

Next, the CR4 MSRs, `IA32_VMX_CR4_FIXED0` and `IA32_VMX_CR4_FIXED1`, are read and CR4's bits are set according to the default values reported by these MSRs. The return values for `IA32_VMX_CR4_FIXED0` and `IA32_VMX_CR4_FIXED1` for the target processor in this work are shown in Figure 15.

IA32_VMX_CR4_FIXED0- EAX: 0x2000	31	1513	0
	0000	0000 0000 0000 0010	0000 0000 0000
IA32_VMX_CR4_FIXED1- EAX: 0x67ff	31	1513	0
	0000	0000 0000 0000 0110	0111 1111 1111
Required Values for CR4	31	1513	0
	0000	0000 0000 0000 010	0

Figure 15. MSRs showing CR4 required values

The CR4.VMXE (bit 13) is the only value that must be set. This bit is the VMX enable bit and enables VMX operation when set. Bits 31:15 and bits 12:11 must be clear while, bit 14, and bits 10:0 are flexible and may be set if the functionality (such as page size extensions (PSE), physical address extension (PAE), enabling safer mode extensions (SMX), etc.) is desired for the VMM.

Finally, the `IA_32_FEATURE_CONTROL` MSR is read. Although other VMX MSRs are read-only and will cause a general-protection exception (GP(0)) on any attempt to write to them, the `IA_32_FEATURE_CONTROL` MSR is used in a similar fashion as



the control register CR4; the bits of this register are set or cleared to control the functionality of VMX operation. The relevant bits of this MSR are described as follows:

- **Bit 0 is the lock bit.** If this bit is clear, VMXON causes a general-protection exception. If the lock bit is set, WRMSR to this MSR causes a general-protection exception; the MSR cannot be modified until a power-up reset condition. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX. To enable VMX support in a platform, BIOS must set bit 1, bit 2, or both (see below), as well as the lock bit. [36]
- **Bit 1 enables VMXON in SMX operation.** Attempts to set this bit on logical processors that do not support both VMX operation and SMX operation cause general-protection exceptions. [40]
- **Bit 2 enables VMXON outside SMX operation.** Attempts to set this bit on logical processors that do not support VMX operation cause general-protection exceptions. [40]

As with the CR4.VMXE bit, bit 0 of the IA\_32\_FEATURE\_CONTROL MSR must be set in order to enter VMX operation. The SMX, referred to in the description of bit 0, stands for Safer Mode Extensions and enables using a trusted platform module [37], i.e., similar to that described earlier in the Terra project. SMX operation is not being used for the purposes of this effort so bit 1 is left clear.

To enable VMXON without using SMX it is necessary to set bit 2 of the IA\_32\_FEATURE\_CONTROL MSR. The only problem with doing this was that it was assumed that these bits would be clear and that the `setup_vmxon()` function would need to set bits 0 and 2 to enable VMXON. Attempts to set these bits would cause a general-protection exception. Upon printing the value in this MSR, it was realized that these bits were being set by the BIOS. This would have been fine, but they were set for SMX operation and this was not desired. As described above, once the lock bit (bit 0) is set it is not possible to change this MSR. So this had to be changed in BIOS for the target platform. For this platform, the setting was found enumerated in the BIOS user interface under the heading “Performance.” The name of the field that had to be changed was named “Trusted Execution” in the BIOS user interface. Once this was set to “off”, the

BIOS set bits 0 and 2 of the IA\_32\_FEATURE\_CONTROL MSR as required for the testing being done with the LPSK. The section of the setup\_vmxon() function that sets the IA\_32\_FEATURE\_CONTROL MSR bits was changed to check if the lock bit had already been set by BIOS before attempting to set any of the other bits. If the lock bit is set, then the function determines if the bits are already set correctly with bit 2 and bit 0 set. If the bits are set correctly, then the function will call VMXON. If the bits are set incorrectly, then the function halts the processor.

With the setup necessary for the system to transition to VMX root operation complete, all that remains is to execute the VMXON instruction As follows: “Execute VMXON with the physical address of the VMXON region as the operand” [38].

A search of the documentation and code for the Watcom [39] compiler that is used for the LPSK development showed that Watcom does not implement any of the VMX instructions, so it was necessary to write a pragma to call VMXON. A *pragma* is a compiler directive that, for our purposes, can be used to direct the compiler to emit the operation code (opcode) for calling VMXON.

Originally, this pragma was created with no calling parameters. The address of the VMXON region was provided in a variable that was moved to the EBX register (using MOV EBX) within the pragma before the opcode for the VMXON instruction. The pragma was tested by calling the vmxon pragma with a parameter that has been assigned the address of the VMXON region. The pragma moves the parameter value to the EBX register and then provides the opcode that the compiler must emit to call the VMXON instruction. Calling the vmxon pragma with this parameter value caused a general-protection exception (GP()). The Intel documentation for the VMXON instruction [40] provided some clues as to what could be causing the general-protection exception. Intel provides the logic for the VMXON instruction in pseudo code. Only one section of this pseudo code could cause the GP(). As shown in Figure 16.

```

IF (CPL > 0) or (in A20M mode) or
(the values of CR0 and CR4 are not supported in VMX operation2) or
(bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
(in SMX operation3 and bit 1 of IA32_FEATURE_CONTROL MSR is clear) or
(outside SMX operation and bit 2 of IA32_FEATURE_CONTROL MSR is clear)
THEN #GP(0);

```

Figure 16. VMXON pseudo code. From [44]

The first thing that stood out was the A20M mode. The `setup_vmxon()` function did nothing with the A20 bit and there was no means to access it in the LPSK code. A function was created to get and set this bit. Changing the setting of this bit did not change the `GP()` received each time the `vmxon` pragma was called. (It was only after VMXON was working that it was determined that the A20 bit was not influential.) Next, each of the CR0, CR4, and IA32\_FEATURE\_CONTROL MSR were printed out and examined to make sure they were set properly. Each appeared to be set according to the Intel specification for executing the VMXON instruction. Some further information about why VMXON might cause a `GP()` was given in a note below the pseudo code, as shown in Figure 17.

#### Protected Mode Exceptions

#GP(0)	If executed outside VMX operation with <code>CPL&gt;0</code> or with invalid CR0 or CR4 fixed bits. If executed in A20M mode. If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains an unusable segment. If the source operand is located in an execute-only code segment.
--------	--

Figure 17. Further explanation of VMXON `GP()` exceptions. From [44]

This section gave a requirement that had not been checked: the source operand to VMXON must be for a segment referenced by any of the segment registers. To determine if this was the cause of the `GP()`, the selector returned when the VMXON

region was added to the GDT, was copied into the GS register. This way, it was certain that one of the segment registers would definitely refer to the segment that the VMXON region was within when VMXON was called.

Unfortunately, this produced the same GP() exception. However, another reference to the requirements to the operand (or parameter) of VMXON stated,

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory. [44]

This reference to a pointer seemed to indicate that the VMXON parameter may not actually be the address of the VMXON region, but a reference to where the address is stored in memory.

A question posted to the Intel Forum returned the following response, “What VMXON takes is a reference to a memory location that always contains a “64 bit physical” address (regardless of which mode you are running in) which in turn points to the VMXON region” [41]. With this as a guide, a 64-bit location in memory was created and this location was assigned the physical address of the VMXON region. The parameter passed to VMXON was assigned the address of the 64 bit location in memory. See Figure 18.

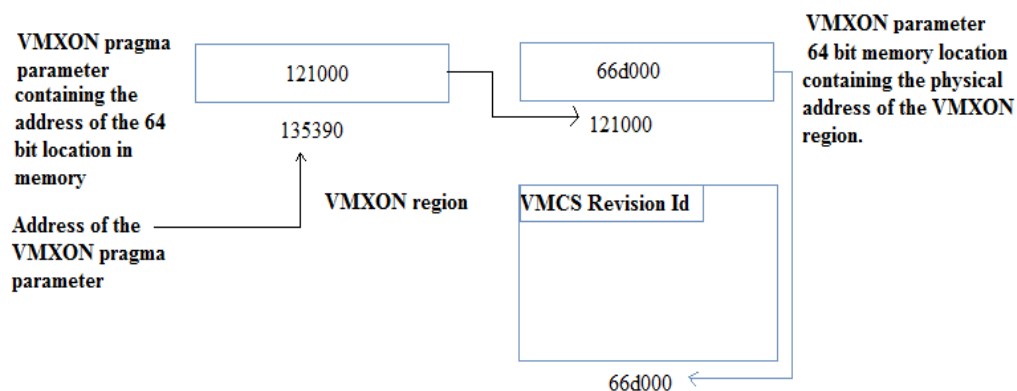


Figure 18. VMXON parameter

Setting the VMXON parameter as described above and calling VMXON still gave a GP(). On a hunch that the processor may require that the VMXON region be referenced by the DS register, the VMXON region was allocated in the segment referenced by the DS segment register. Trying this still produced a GP(). Following this train of thought further, it was determined to declare a variable to be used as the VMXON parameter. This variable's physical address will be within the segment referenced by the DS register. Calling VMXON with the VMXON parameter and the VMXON region both in the segment pointed to by the DS segment register did finally transition the processor to VMX root operation. To ensure that VMXON successfully transitioned the processor to VMX root operation, the EFLAGS.CF (bit 0) was printed to the display. Upon successful execution of the VMXON instruction, this bit is set to zero. As a further test to ensure successful transition to VMX root operation, the VMXOFF instruction was called. This instruction will only execute if the processor is running in VMX root operation. If this instruction is called outside of VMX root operation, then it will cause an undefined opcode exception (UD) [42].

To ensure that executing VMXOFF produced the expected UD exception when executed outside of VMX root operation, VMXOFF was executed before transitioning to VMX root operation and did produce a UD exception. After this test VMXOFF was executed after the processor had transitioned into VMX root operation. This time, VMXOFF executed properly (no exceptions) and the LPSK could now be successfully transitioned in and out of VMX root operation.

It did not seem right that the VMXON region had to be in the segment referred to by the DS segment register. So, tests were performed to see if the VMXON region could be allocated in other segments. The first test placed the VMXON region in its own segment and set the DS segment register to refer to this segment. The parameter to VMXON was left in the kernel data segment with the FS segment register referring to the kernel data segment where this parameter was allocated. Calling the VMXON instruction this way caused a GP(). So, it is evident that the VMXON parameter must be referred to by the DS register, but the question remains as to whether the VMXON region can be in a segment referred to by a segment register other than DS.

A second test was performed to determine whether VMXON would work with the VMXON region in its own segment. For this test, the DS segment register was set to refer to the kernel data segment where the VMXON parameter was and the GS segment register was set to refer to the segment where the VMXON region was allocated. With the VMXON region and VMXON parameter referred to by two different selectors, it was possible to properly transition the LPSK into and out of VMX root operation. Table 1 outlines each test and its outcome.

Table 1. Trials and outcomes for testing VMXON

VMXON instruction	VMXON region	VMXON parameter	DS selector	GS selector	FS Selector
Works	Allocated in kernel data segment	Allocated in kernel data segment	Refers to kernel data segment	Refers to kernel data segment	Refers to kernel data segment
Does not Work	Allocated in its own segment	Allocated in kernel data segment	Refers to segment where VMXON region is allocated	Refers to kernel data segment	Refers to kernel data segment
Works	Allocated in its own segment	Allocated in kernel data segment	Refers to kernel data segment	Refers to segment where VMXON region is allocated	Refers to kernel data segment

## IV. VIRTUAL MACHINE CONTROL STRUCTURE ANALYSIS

A Virtual Machine Control Structure (VMCS) is a 4KB aligned region (VMCS region) in memory that a logical processor uses to store processor state information (control registers, segment registers, rflags, etc.), for both the guest and host during VMX operation. The size and format of the VMCS is processor-specific and can be determined from the IA\_32\_BASIC MSR. Bits 31:0 of this MSR are termed the VMCS revision identifier and are used to identify the specific format that a given processor uses for the data area of the VMCS. The first 4 bytes of the VMCS must contain this VMCS revision identifier. The next 4 bytes are reserved for the processor to store information about the cause of VMX aborts. The remaining bytes are termed the VMCS data area and their format is determined by the processor. The format of the VMCS region is shown in Figure 19.

Byte Offset	Contents
0	VMCS revision identifier
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

Figure 19. VMCS region format. From [43]

The general layout of the VMCS data area can be broken up into six areas as follows:

- The Guest-State area.
- The Host-State area.
- The VM-execution Control fields.
- The VM-exit Control fields.
- The VM-entry Control fields.
- The VM-exit Information fields

The fields of the VMCS data area can be accessed using the VMREAD and VMWRITE instructions. Each field of a VMCS is associated with a 32-bit value called an encoding.

This encoding is provided to the VMREAD and VMWRITE instructions to read or update a field in the VMCS, respectively. The layout of the 32-bit encoding is shown in Figure 20.

Bit Position(s)	Contents
31:15	Reserved (must be 0)
14:13	Width: 0: 16-bit 1: 64-bit 2: 32-bit 3: natural-width
12	Reserved (must be 0)
11:10	Type: 0: control 1: read-only data 2: guest state 3: host state
9:1	Index
0	Access type (0 = full; 1 = high); must be full for 16-bit, 32-bit, and natural-width fields

Figure 20. VMCS component encoding structure. From [43]

The width field tells the width of the field as shown. Natural-width is 64 bits on processors that support Intel 64-bit architecture and 32-bit otherwise. The index field is used to distinguish between fields that have the same width and type.

This information is used to manage transitions to and from VMX non-root operation, and to manage processor behavior while in VMX non-root operation (for example, what causes a VM to exit?). A VMM uses a separate VMCS to manage each VM that it provides resources for. As shown in Figure 21, a VM-entry is a transition from the VMM in VMX root operation to a VM in VMX non-root operation, and a VM-exit is a transition from a VM in non-root operation to the VMM in VMX root operation.



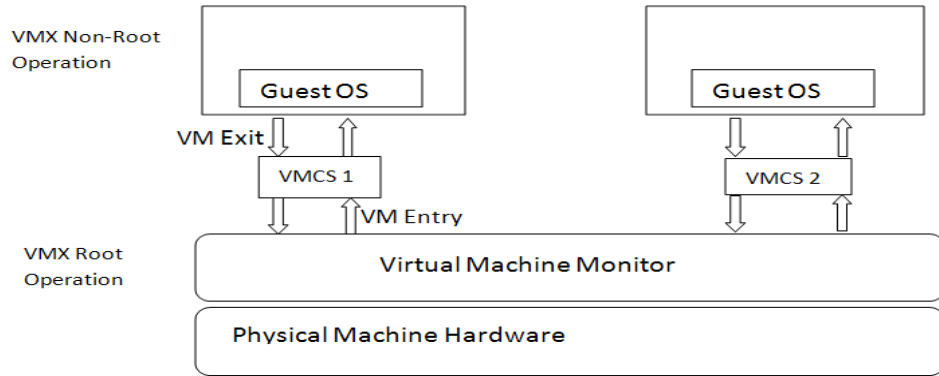


Figure 21. VM-entry/VM-exit

On a VM-exit, a processor will store information about the cause of the VM-exit in the VM-exit information fields, store the guest state information, and load the host state from the current VMCS. A processor can maintain multiple active VMCS, but only one VMCS can be current. The VMLAUNCH, VMRESUME, VMREAD and VMWRITE instructions only operate on the current VMCS [43]. A VMCS is made current by executing the VMPTRLD instruction. This instruction makes a VMCS active and current and provides a pointer to this VMCS for the processor to use to read and update the VMCS. Some of the processor state loaded on VM-exit is based on the VM-exit control fields of the VMCS.

A VM-entry can be accomplished by software executing the VMLAUNCH or VMRESUME instructions. Each VMCS has a launch-state field that is used to determine which VM-entry instruction is used. VMLAUNCH requires a VMCS that has the launch-state field clear, while the VMRESUME instruction requires a VMCS that has the launch-state field set. When either the VMLAUNCH or VMRESUME instruction is executed, the processor first checks to see if there is a current VMCS. The processor then checks the fields of the control and host state areas of the current VMCS to ensure they are properly set to support VMX non-root operation.

Once the fields of the control and host state areas of the VMCS are checked, the processor begins to check and load the guest state area of the VMCS. Loading the guest state entails loading some of the state information of the processor, such as CR0, CR3 and segment registers, from the corresponding fields in the VMCS. Other state

information is loaded based on the control fields of the VMCS. Checking the fields of the guest state area of the VMCS includes checking (1) that none of the control registers set bits that are not supported in VMX operation, (2) the access right fields of the segment registers, and (3) the GDTR and IDTR fields. If all checks of the VM-entry succeed and all state information loaded to the processor is correct for the Intel IA 32 architecture (this is checked during the VM-entry process), then transitioning to VMX non-root operation is allowed to proceed. In this mode, the guest OS stack is associated with and managed by a logical processor. Operation of the processor in this mode is similar to that outside a virtualized environment.

The main difference between VMX non-root operation and ordinary processor operation is the instructions and memory accesses that cause VM-exits. VM-exits are fault-like, in that the instruction causing the VM-exit is not executed and processor state is not updated. Instead the instruction is handled by the VMM. Certain instructions, such as INVD, CPUID, and all the VMX instructions, will always cause VM-exits. Other events, such as task switches, will also cause VM-exits, as they are not allowed in VMX non-root operation. Using the control fields in the VMCS, the VMM software can control the exceptions (faults, interrupts and aborts) that will cause VM-exits.

## **V. CONCLUSIONS AND FUTURE WORK**

### **A. CONCLUSIONS**

This work has shown that the LPSK can be modified to run in Intel VMX root operation. Several changes were necessary to accommodate this. A function to set up and enable paging had to be added to the LPSK. It was also necessary to add get and set functions to the LPSK to accommodate changing the necessary control registers and accessing model specific registers (MSRs) of the processor. These functions were used to determine the processor-specific setup requirements for VMX root operation. Using the get and set functions, a function was constructed that read the necessary CPU information and set up necessary control structures in memory and properly set control registers and model specific registers of the test hardware.

Finally, the compiler for this project did not have VMX instructions, so a compiler pragma was added that allowed the compiler to output the operation code necessary to call the VMXON instruction.

Tests were necessary to determine the exact parameter requirements of the VMXON instruction. The work required included:

- Determination of the required mode of operation for VMXON, i.e., protected mode with paging enabled.
- Determination of the paging mode that would work for the LPSK, i.e., 32-bit paging mode.
- Determination of the required settings for the paging structure entries.
- Changing the task set up code of the LPSK to allow paging to function properly.
- Determination of the control register settings to enable paging.
- Creation of a function to set up and initialize the entries of the paging structures.

- Tests of the paging function.
- Determination of the steps necessary to prepare the CPU to operate in VMX root operation.
- Determination of how to read the CPU specific requirements for VMX root operation.
- Addition of functions to the LPSK to access CPU-specific processor information, i.e., CPUID and MSRs.
- Creation of a function to set up the necessary control structures for VMX operation (VMXON region and VMCS region), and read the CPU-specific requirements for the size and structure of the control structures. This function also read the CPU-specific requirements for the control registers and MSRs related to VMX root operation, and ensured that these control registers and MSRs were set according to the CPU-specific requirements of the target processor.
- Addition of pragmas were added to allow the compiler to produce the operation code necessary to execute the VMXON and VMXOFF instructions.
- Tests to determine the requirements of the VMXON instruction.
- Tests to ensure that the processor was operating in VMX root operation.

With the outlined research, testing and modification, the LPSK ran in VMX root operation.

## **B. FUTURE WORK**

The immediate follow-on work that should be performed is an analysis of the minimum requirements to setup the VMCS in order to launch a guest VM. Some analysis of the VMCS structure has been performed and has been provided. Further analysis will be necessary to determine the proper VMCS register settings and control settings required to operate the LPSK as a guest VM. One item of interest will be

considering that in VMX non-root operation a task switch causes a VM-exit. This means the LPSK would require modification if executed in a VM. Another item to consider will be whether to set up the LPSK as an unrestricted guest and what that might mean for the operation of the LPSK as a VM. Setting the VM-execution control to allow unrestricted guest operation of a VM allows a VM to launch without paging and protection enabled, and can also allow the VM to operate in real-address mode.

Future work may also include an analysis of the VMCS settings necessary to launch other OSs as VMs to an LPSK-based VMM. Once VMs based on multiple OSs are operating, an analysis of the security provided by an LPSK-based VMM operating multiple VMs in isolated partitions would be possible. It would also be interesting to determine if a resource such as a network connection could be securely shared among VMs running on an LPSK-based VMM. Finally, work could be done to determine any VM-execution and VM-exit settings of the VM environments that will maintain security and make possible enhancements to the speed of the operation of the VMs in this environment.

THIS PAGE INTENTIONALLY LEFT BLANK

## APPENDIX A

This appendix provides a copy of the code required to enable paging for the LPSK. Code comments are provided in an effort to make the code self explanatory.

### Paging function:

```
/* setup paging function */

void setup_paging(void)
{
/*
*   This function performs the necessary steps
*   to setup and enable paging.
*
*   First a page directory structure is allocated
*   in physical memory. Second, 1024 page table
*   structures are allocated. The size of each of these structures
*   (page table and page directory) is 4KB,
*   and the physical address of each is aligned on 4K.
*   boundaries in physical memory. (One
*   contiguous block of memory is allocated for all 1024
*   page table regions so that only one selector
*   to this contiguous block in memory
*   is created in the GDT.)
*
*   We are enabling 32 bit paging-
*   mode, so each entry in the paging
*   structures is 32 bits. Each entry must
*   contain a physical address that references
*   either, another paging structure, or references
*   the physical address of the region in memory to
*   which the linear address refers.
*
*   For the page directory table, each entry references
*   a 4KB page table structure. (For 32 bit paging,
*   the physical address to which the entry refers is placed
*   in bits 31:12 of each entry.) To set bits 31:12
*   of each page directory entry to the physical address
*   of the start of a page table structure, a loop
*   is provided that starts at the first address
*   of the page table structure that was allocated,
*   and steps (or adds) 4k to that address and sets
*   the next entry's bits 31:12 to this address...
*   This loop also sets three flag bits for each entry: present
*   (bit 0), read/write (bit 1) and user/supervisor (bit 2) flag
*   bits for each entry. Then the page table entries
*   are set in a similar fashion, but this time each
*   entry in the page table references a 4k page frame
```

```

*   in physical memory.
*
*   Next the control registers are set to enable 32 bit
*   paging.
*
*   CR4 page size extensions (bit 4), physical
*   address extension(bit 5), and page global
*   enable (bit 7) are cleared (set to 0).
*
*   CR3 bits 31:12 are set equal to the beginning
*   address of the page directory structure.
*
*   CR0 protection enable (bit0), write protect (bit 16)
*   paging (bit 31) are set
*
*/
/* Declare variables */
int idx, idy, result, pgdir_size, pgtbl_size;
unsigned int *pgdir_ptr, *pgtbl_ptr;
unsigned int pgdir_addr, pgtbl_addr, curr_addr, ctrl_reg;
unsigned short pgdir_sel, pgtbl_sel;

/* Create the page directory region
 * We need a selector to it, so we can write to it
 * Size is 4KB, alignment is 4K
 */
pgdir_size = 0x1000;
result = kmem_allocate(pgdir_size, 0x1000, &pgdir_addr);
if (!result)
{
    result = add_gdt_data_segment(pgdir_addr, pgdir_size, 0,
                                RW, &pgdir_sel);
}
if (result)
{
    printf_int("Can't add page directory region %d\n", result);
    halt(); /* Never returns */
}
pgdir_ptr = make_ptr(pgdir_sel, 0);

/* Create the page table region
 * We need a selector to it, so we can write to it
 * We need a total of 1024 4KB page tables, but don't
 * want to make that many entries
 * in the GDT, so just make 1 segment to hold
 * all the 4KB page table regions.
 * Size of the segment containing the page table
 * entries is 4KB * 1024, alignment is 4K.
 */
pgtbl_size = 0x1000 * 1024;
result = kmem_allocate(pgtbl_size, 0x1000, &pgtbl_addr);
if (!result)
{

```



```

        result = add_gdt_data_segment(pgtbl_addr, pgtbl_size, 0,
                                      RW, &pgtbl_sel);
    }
    if (result)
    {
        printf_int("Can't add page table region %d\n", result);
        chalt(); /* Never returns */
    }
    pgtbl_ptr = make_ptr(pgtbl_sel, 0);

    /* Now fill in the page table.
     * Each entry of the page table references a 4KB page
     * frame, the physical address of the page frame to which
     * the entry refers is placed in bits 31:12 of each entry. Also, for
     * each entry we set the present (bit 0), read/write (bit 1) and
     * user/supervisor (bit 2) flag bits.
     */

    curr_addr = 0x0;
    for (idx=0; idx < 1024; idx++)
    {
        for (idy=0; idy < 1024; idy++)
        {
            /* This sets the page base address */
            pgtbl_ptr[(idx*1024) + idy] = curr_addr;
            /* Now set bit fields */
            pgtbl_ptr[(idx*1024) + idy] |= 0x7; /* Present bit */
            /* Setup for next page address */
            curr_addr += 0x1000;
        }
    }

    /* Now fill in the directory table.
     * Each entry of the directory table references a 1k page
     * table structure, the physical address of the page table to which
     * the entry refers is placed in bits 31:12 of each entry. Also, for
     * each entry we set the present (bit 0), read/write (bit 1) and
     * user/supervisor (bit 2) flag bits.
     */

    curr_addr = pgtbl_addr;
    for (idx=0; idx < 1024; idx++)
    {
        /* This sets the page table base address */
        pgdir_ptr[idx] = curr_addr;
        /* Now set bit fields */
        pgdir_ptr[idx] |= 0x7; /* Present bit */
        /* Setup for next page table address */
        curr_addr += 0x1000;
    }

    /* make sure PSE, PAE and PGE bits in cr4 are clear */

    ctrl_reg = get_cr4();
    /* if CR4.PSE (page size extensions- bit 4) is set-

```

```

    * clear the bit
    */

    if ((ctrl_reg & 0x10) != 0) ctrl_reg ^= 0x10;

    /* if CR4.PAE (physical address extension- bit 5) is set-
    * clear the bit
    */
    if ((ctrl_reg & 0x20) != 0) ctrl_reg ^= 0x20;

    /* if CR4.PGE (page global enable- bit 7) is set-
    * clear the bit
    */
    if ((ctrl_reg & 0x80) != 0) ctrl_reg ^= 0x80;
    set_cr4(ctrl_reg);

/* Set the page directory address in CR3 */

/* set address of page directory in bits 31:12 of CR3 */
set_cr3(pgdir_addr);
ctrl_reg = get_cr3();

/* Enable paging, set PE, WP and PG bits in CR0 */

ctrl_reg = get_cr0();

/* set CR0.PE (protection enable bit- 0) if it is not set */
if ((ctrl_reg & 0x1) == 0) ctrl_reg |= 0x1;
/* set CR0.WP (write protect- bit 16) */
ctrl_reg |= 0x10000;
/* set CR0.PG (paging- bit 31) */
ctrl_reg |= 0x80000000;
set_cr0(ctrl_reg);
} /* setup_paging */

```

## APPENDIX B

This appendix provides a copy of the code required to transition the target processor of the LPSK into VMX root operation. Code comments are provided in an effort to make the code self explanatory

### VMXON

```
/* setup VMXON
*
*
* This function performs the tasks necessary
* to transition the LPSK into VMX root
* operation.
*
* The parameter to this function (param_sel)
* provides the selector to the segment in memory
* that contains the VMXON parameter.
*
* First, all the model
* specific registers (MSRs) are declared
* as constants to make it clear which
* is being referenced with each call. Next,
* the CPU identification (CPUID) is called
* to determine if the processor supports
* VMX operation and the accompanying VMX
* related MSRs. IF VMX operation is supported
* then the MSRs are read to determine
* the processor specific size requirements
* for the VMXON region and the VMCS region. The
* VMCS revision identifier, which provides the
* processor specific requirements for the data
* portion of the VMCS, is read from the MSR as well
*
* A region in memory is allocated to be used for the
* VMXON region. This region is added to the GDT to
* obtain a selector for this region. A pointer is
* made so this region can be accessed.  *
* The necessary MSRs are read to
* determine the processor specific
* settings required for CR0, CR4 and the
* IA_32_FEATURE_CONTROL MSR. Once the
* settings are correct VMXON is called to
* transition the system to VMX root
* operation.
*
* Note: location of functions used.
* printf- Utility module.
* kmem_allocate- Memory management module.
* add_gdt_data_segment - GDT module.
```

```

*  chalt- Utility module.
* */

void setup_vmxon (unsigned short param_sel)
{
/* Variables for cpuid read function */
    int cpuid_eax, cpuid_ebx, cpuid_ecx cpuid_edx;

/* Variables for rdmsr read function */
    int msr_eax, msr_edx;

/* Function variables */
    int            results, region_size, vmcs_revision_id, idx;
    unsigned int   *vmxon_ptr;
    unsigned int   vmxon_addr, ctrl_reg;
    unsigned int   eflags;
    unsigned short vmxon_sel;
    unsigned long long *address_of_vmxon_param;

/* Assign constant values for the model specific registers (MSRs)
* these registers have constant locations
*/
    #define IA32_VMX_BASIC_MSR 0x480
    #define IA32_VMX_CR0_FIXED0_MSR 0x486
    #define IA32_VMX_CR0_FIXED1_MSR 0x487
    #define IA32_VMX_CR4_FIXED0_MSR 0x488
    #define IA32_VMX_CR4_FIXED1_MSR 0x489
    #define IA32_FEATURE_CONTROL_MSR 0x3A

/* check cpuid to see if vmx supported on this cpu */
/* parameters for get_cpuid- first two are calling parameters
* that will be placed in EAX and ECX respectively,
* the last four are variables for the return values
* of this function
*/
    get_cpuid(0x1,0x0, &cpuid_eax, &cpuid_ebx, &cpuid_ecx,
    &cpuid_edx);

/* bit 5 not set indicates no vmx support */
    if((cpuid_ecx & 0x20) == 0) {
        printf_lstr("VMX not supported on this CPU!");
        chalt(); /*Never returns*/
    }
/* bit 5 not set indicates no support for model
* specific registers (MSRs)
*/
    if((cpuid_edx & 0x20) == 0)

    {
        printf_lstr("MSR not supported on this CPU!");
        chalt(); /*Never returns*/
    }

/* Read IA32_VMX_BASIC_MSR to obtain processor
* requirements for VMXON region and VMCS region

```

```

* (both regions will be the same size)
*/
    /* get IA32_VMX_BASIC MSR */
    get_msr(IA32_VMX_BASIC_MSR, &msr_eax, &msr_edx);

    /* get value in first 12 bits of this register */
    msr_edx &= 0xfff;

    /* bits 12:0 (44:32 of MSR) is the number of bytes that should be
     * allocated for the VMXON & VMCS regions
     */
    region_size = msr_edx;

    /* bits 31:0 are vmcs revision identifier for this cpu */
    vmcs_revision_id = msr_eax;

/* Allocate a region in memory of size "region_size"
 * aligned at a 4k boundary*
*/
    /* allocate a region in memory of size "region_size" with
     * physical address aligned at a 4k boundary.
     * Parameters of kmem_allocate are as follows:
     * size in bytes, boundary offset, variable to
     * return address to region once assigned.
     */
    results = kmem_allocate(region_size, 0x1000, &vmxon_addr);
    if (!results)
    {
        /* parameters to add_gdt_data_segment are as follows:
         * beginning address of the region to be added, size of the
         * segment, highest privilege level that can access this
         * segment, access rights, variable to assign the selector
         * to this segment once it is assigned.
         */
        results = add_gdt_data_segment(vmxon_addr, region_size, 0,
            RW, &vmxon_sel);
    }
    if (results)
    {
        printf_int("Can't add vmxon region to GDT %d\n",
            results);
        chalt(); /*never returns*/
    }

/* make a pointer to reference the VMXON region */
    vmxon_ptr = make_ptr(vmxon_sel, 0);

/* Set bits 31:0 of VMXON region to the vmcs revision id obtained from
 * IA32_BASIC_MSR
 */
    vmxon_ptr[0] = vmcs_revision_id;

/* setup CR0 fixed bits according to MSRs */

```

```

/* get IA32_VMX_CR0_Fixed0 MSR */
get_msr(IA32_VMX_CR0_FIXED0_MSR, &msr_eax, &msr_edx);
ctrl_reg = get_cr0();

/* any bit set to 1 in IA32_VMX_CR0_FIXED0_MSR should be set to 1
 * in CR0
 */
ctrl_reg |= msr_eax;

/* get IA32_VMX_CR0_Fixed1 MSR */
get_msr(IA32_VMX_CR0_FIXED1_MSR, &msr_eax, &msr_edx);

/* any bit set to 0 in IA32_VMX_CR0_FIXED1_MSR should be 0 in CR0
 * (other bits flexible)
 */
ctrl_reg &= msr_eax;
set_cr0(ctrl_reg);

/* setup CR4 fixed bits according to MSRs */
/* get IA32_VMX_CR4_Fixed0 MSR */
get_msr(IA32_VMX_CR4_FIXED0_MSR, &msr_eax, &msr_edx);
ctrl_reg = get_cr4();

/* any bit set to 1 IA32_VMX_CR4_FIXED0_MSR should be set to 1 in
CR0 */
ctrl_reg |= msr_eax;

/* get IA32_VMX_CR4_Fixed1 MSR */
get_msr(IA32_VMX_CR4_FIXED1_MSR, &msr_eax, &msr_edx);

/* any bit set to 0 in IA32_VMX_CR4_FIXED1_MSR should be 0 in CR0
 * (other bits flexible)
 */
ctrl_reg &= msr_eax;
set_cr4(ctrl_reg);

/* set bits in IA32_Feature_Control MSR */
get_msr(IA32_FEATURE_CONTROL_MSR, &msr_eax, &msr_edx);

/* bit 6 not set in IA32_FEATURE_CONTROL_MSR (cpuid_ecx)
 * no smx support (we will not use smx or
 * measured VMM at this point)
 */
if((cpuid_ecx & 0x40) == 0)
{
    printf_lstr("SMX not supported on this CPU \n");
    pause("pause for smx not supported");
    /* This version of VMXON does not
     * use SMX so code can continue
     * If SMX is required in the future code will go here
     */
}

/* If msr_eax bit 0 is set- then the lock bit is set-
 * do not try to set or will get GP exception

```

```

    */
    if((msr_eax & 0x1) == 1) {
        /* bits 2 and 0 set incorrectly so halt*/
        if((msr_eax & 0x5) != 0x5) {
            printf_int("IA32_FEATURE_CONTROL_MSR is
            set incorrectly it is equal to... 0x%x\n", msr_eax);
            chalt(); /* Never returns */
        }
    }
    else /* bit 0 (lock bit) is not set,
        * so we can write to the IA32_FEATURE_CONTROL_MSR
        */
    {
        /* make sure bit 1 in MSR is 0
        *(we do not want SMX operation at this time)
        */
        msr_eax &= 0x5;
        /* set bit 0 and bit 2 in MSR (bit 0 is vmxon lock bit and
        * bit 2 enables vmxon outside SMX)
        */
        msr_eax |= 0x5;
        set_msr(IA32_FEATURE_CONTROL_MSR, msr_edx, msr_eax);
    }

    /* Set parameter address for parameter to VMXON */
    /* get_vmxon_addr() returns a physical address of a 64 bit region
    * of memory in the kernel data segment
    */
    address_of_vmxon_param = (unsigned long long *) get_vmxon_addr();

    /* assign the physical address of VMXON region to the 64 bit location
    * in memory that address_of_vmxon_param points to
    */
    *address_of_vmxon_param = vmxon_addr;

    /* set gs segment to refer to the segment that vmxon region is within
    */
    set_gs(vmxon_sel);

    /* execute VMXON */
    /*The parameters to the vmxon pragma are a physical address
    * of a 64 bit region in memory (address_of_vmxon_param),
    * this 64 bit region in memory is the parameter to VMXON,
    * and the selector (param_sel) of the segment that 64 bit
    * region is in.*/
    vmxon(address_of_vmxon_param, param_sel);

    printf_int("eflags after first VMXON = 0x%x\n", get_flags());
    pause(" for eflags after vmxon");

    /* execute VMXOFF- this should only succeed if we are in VMM root
    * operation
    */
    vmxoff();

```

```
/* execute VMXON */
vmxon(address_of_vmxon_param,param_sel);

    printf_int("eflags after second VMXON = 0x%x\n", get_flags());
    pause(" for eflags after vmxon");

}/* setup VMXON */
```



## APPENDIX C

This appendix provides copies of the pragmas created to allow the Watcom compiler to emit the code necessary to call the VMXON function.

Pragmas:

```
/* VMXON
 *
 *This pragma is necessary because the Watcom compiler
 * does not support VMX instructions. This pragma
 * executes the VMXON instruction
 *
 *
 * The addr parameter is the address of the
 * parameter to the VMXON opcode.
 * the address parameter is passed in the EBX
 * register. This is the offset in the segment.
 * The sel parameter is the selector of the segment
 * containing the parameter to the VMXON opcode,
 * this parameter is passed in the CX register.
 * Thus, the selector and offset provide a full
 * pointer to the opcode.
 * The VMXON opcode expects DS to refer to the
 * segment containing its address parameter.
 *
 * 0xF3 0x0F 0xC7 are the VMXON opcode bytes
 *
 * 0x33 completes the VMXON opcode and is its modifier byte *      this
modifier tells the processor that the parameter
 * to the VMXON opcode is in EBX
 *
 */
```

```
void __near vmxon(unsigned int addr, unsigned short sel);
#pragma aux vmxon = \

    "mov ds, cx" \
    0xF3 0x0F 0xC7 0x33 \
    parm [ebx][cx];
```

```
/* VMXOFF
 *
 * This pragma is necessary because the Watcom compiler
 * does not support VMX instructions. This pragma
 * executes the VMXOFF instruction.
 *
 * 0x0F 0x01 0xC4 are the VMXOFF opcode bytes.
 *
 */

void __near vmxoff(void);
#pragma aux vmxoff = \
    0x0F 0x01 0xC4
```

## LIST OF REFERENCES

- [1] R. P. Goldberg, “Architectural principles for virtual computer systems,” PhD dissertation, Harvard University, 1972.
- [2] Information Assurance Directorate, National Security Agency, Fort George G. Meade, MD. “U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness,” Version 1.03, June 2007.
- [3] T. E. Levin, C. E. Irvine, and T. D. Nguyen, “Least privilege in separation kernels,” *Proc. International Conference on Security and Cryptography*, Setúbal, Portugal, August 2006, pp. 355–362.
- [4] R. J. Creasy, “The origin of the VM/370 time-sharing system,” *IBM Journal of Research and Development*, vol. 25, no.5, pp.483–490, Sep. 1981. doi: 10.1147/rd.255.0483.
- [5] R. P. Parmelee, T. I. Peterson, C. C. Tillman, D. J. Hatfield, “Virtual storage and virtual machine concepts,” *IBM Systems Journal*, vol. 11 no. 2 pp. 99–130, 1972.
- [6] R. P. Goldberg, “Survey of Virtual Machine Research,” *Computer*, vol. 7 no. 6, pp. 34–45, June 1974.
- [7] IBM, System/370 model 145 [Online]. Available: [http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe\\_PP3145.html](http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP3145.html) [Accessed: Aug. 5, 2010].
- [8] IBM, Z/VM the newest VM hypervisor based on 64 bit Z architecture [Online]. Available: <http://www.vm.ibm.com/> [Accessed: Aug. 5, 2010].
- [9] Microsoft, Hyper-V Server 2008 R2, [Online]. Available : <http://www.microsoft.com/hyper-v-server/en/us/default.aspx> [Accessed: Aug 26, 2010].
- [10] VMWARE, VMware workstation the gold standard in desktop virtualization [Online]. Available: <http://www.vmware.com/products/workstation/> [Accessed: Aug. 5, 2010].
- [11] B. W. Lampson, “Dynamic protection structures,” *AFIPS Conference Proceedings*, 1969, vol. 35, pp. 27–38.
- [12] S. E. Madnick, and J. J. Donovan, “Application and analysis of the virtual machine approach to information system security,” *Proceedings of ACM SIGARCH-SIGOPS workshop on Virtual computer Systems*, MA, 1973, pp. 210–24.

- [13] R. Rhode, "Secure multilevel virtual computer systems," *Technical Report ESD-TR-74 370*, The MITRE Corporation Bedford: HQ Electronic Systems Division, 1975.
- [14] R. R. GoldLinde, R. J. Peeler, M. Schaefer, J. F. Scheid, and, P. D. Ward, "A security retrofit of the VM/370," *AFIPS Conference Proceedings*, NJ, 1979, vol. 48.
- [15] P. A. Karger, M. E. Zurko, Bonin, W. Douglas, Mason, H. Andrew, Kahn, and E. Clifford, "A VMM security kernel for the VAX architecture," *IEEE*, 1990, PP. 2–19.
- [16] D. T. Bell, LaPadula, and J. Leonard "Computer security model: unified exposition and multics interpretation," Technical Rept. ESD TR-75-306, The MITRE Crop., MA, June 1975.
- [17] K. J. Biba, " Integrity considerations for secure computer systems," Technical Report ESD-TR-76 372, The MITRE Crop., MA, Apr. 1977.
- [18] P. A. Karger, A. J. Herbert, "An augmented capability architecture to support lattice security and traceability of access," *IEEE Symposium on Security and Privacy*, 1984. p. 2.
- [19] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor, " *In Proceedings of the 9<sup>th</sup> USENIX Security Symposium*, Denver, CO, 2000, pp. 129–144.
- [20] T. Grafinkel, B. Pfaff, J. Chow, Rosenblum, Mendel, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing. " *In Proceedings of the 19th Symposium on Operating System Principles (SOSP)*, 2003.
- [21] Trusted Computing Group, TPM main specification [Online]. Available: [http://www.library.dal.ca/Files/How\\_do\\_I/pdf/IEEE\\_Citation\\_Style\\_Guide.pdf](http://www.library.dal.ca/Files/How_do_I/pdf/IEEE_Citation_Style_Guide.pdf) [Accessed: July 30, 2010].
- [22] J. M. Rushby, " Design and verification of secure systems," *ACM Symposium on Operating Systems Principles*, 1981 pp. 12–21.
- [23] W.B. Martin, P.D White, and F.S. Taylor, "Creating high confidence in a separation kernel," *Automated Software Engineering*, 2002, pp. 263–84.
- [24] T. E. Levin, C. E Irvine, and T. D Nguyen, "Least privilege in separation kernels," *Proc. International Conference on Security and Cryptography*, Setúbal, Portugal, 2006, pp. 355–362.

- [25] C. E. Irvine, T. E. Levin, T. D. Nguyen, and G. W. Dinold, "Overview of a high assurance architecture for distributed multilevel security," In *Proceedings Firth IEEE Systems, Man and Cybernetics Information Assurance Workshop*, United States Military Academy, West Point, NY, pp. 10–11.
- [26] CommonCriteriaPortal, "Common criteria for information technology security evaluation" [Online]. Available: <http://www.commoncriteriaportal.org/thecc.html>[Accessed Aug. 2010].
- [27] G. Popek and R. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974. (I-B3).
- [28] Intel Corp., "Intel 64 and IA-32 Architectures Software Developer's Manual," [Online]. vol. 3 p. 2–10. Available: <http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [29] P. Barham, Dragovic, Boris, Fraser, Kier, Hand, Steven, T. Harris, A. Ho, R. Neugebauer, Ian Pratt, and Andrew Warfield, "Xen and the art of virtualization," in *Proceedings 19th ACM Symposium Operating Systems Principles*, ACM Press, 2003.
- [30] Intel Corp., "Intel 64 and IA-32 architectures software developer's manual," [Online]. Vol. 2B, p. 5–1. Available: <http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010].
- [31] R. R Collins, "Detecting intel processors: knowing the generation of a system CPU" [Online]. Available: <http://www.rcollins.org/ddj/Sep96/Sep96.html>. [Accessed Sept. 2010].
- [32] Intel Corp., "Intel 64 and IA-32 architectures software developer's manual," [Online]. Vol. 2A p. 3–197. Available: <http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010].
- [33] Intel Corp., "Intel 64 and IA-32 architectures software developer's manual" [Online]. Vol. 3B p. 20–4. Available: <http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [34] Intel Corp., "Intel 64 and IA-32 architectures software developer's manual" [Online]. Vol. 3B, p. G–3. Available: <http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [35] Intel Corp., "Intel 64 and IA-32 architectures software developer's manual" [Online]. Vol. 3B, Appendix G, p. G–8. Available: <http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].

- [36] Intel Corp., “Intel 64 and IA-32 architectures software developer’s manual” [Online]. Vol. 3B, p. 20–4. Available:  
<http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [37] Intel Corp., “Intel 64 and IA-32 architectures software developer’s manual” [Online]. Vol. 2B p. 6–1. Available:  
<http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [38] Intel Corp., “Intel 64 and IA-32 architectures software developer’s manual” [Online]. Vol. 3B p. 20–4. Available:  
<http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [39] Open Watcom, “Resources” [Online]. Available:  
[http://www.openwatcom.org/index.php/Main\\_Page#Resources](http://www.openwatcom.org/index.php/Main_Page#Resources). [Accessed: Sept 2010].
- [40] Intel Corp., “Intel 64 and IA-32 architectures software developer’s manual,” [Online]. Vol. 2B p. 5–31. Available:  
<http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [41] Intel, “Intel software developer forum” [Online] Available:  
<http://software.intel.com/en-us/forums/showthread.php?t=76617&o=d&s=lr>. [Accessed Aug. 2010].
- [42] Intel Corp., “Intel 64 and IA-32 architectures software developer’s manual,” [Online]. Vol. 2B p. 5–29. Available:  
<http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010 ].
- [43] Intel Corp., “Intel 64 and IA-32 architectures software developer’s manual,” [Online]. Vol. 3B p. 21–1. Available:  
<http://www.intel.com/products/processor/manuals/>. [Accessed Aug. 2010].

## **INITIAL DISTRIBUTION LIST**

1. Defense Technical Information Center  
Ft. Belvoir, VA
2. Dudley Knox Library  
Naval Postgraduate School  
Monterey, CA
3. Dr. Victor Piotrowski  
National Science Foundation  
Arlington, VA
4. Dr. Cynthia E. Irvine  
Naval Postgraduate School  
Monterey, CA
5. David Shifflett  
Naval Postgraduate School  
Monterey, CA
6. Thuy D. Nguyen  
Naval Postgraduate School  
Monterey, CA
7. Jayce Martinsen  
SFS students: Civilian, Naval Postgraduate School)  
Monterey, CA